

1-1-2007

Design implementation and analysis of a dynamic cryptography algorithm with applications

Sourabh Ghose

University of Nevada, Las Vegas

Follow this and additional works at: <https://digitalscholarship.unlv.edu/rtds>

Repository Citation

Ghose, Sourabh, "Design implementation and analysis of a dynamic cryptography algorithm with applications" (2007). *UNLV Retrospective Theses & Dissertations*. 2107.

<http://dx.doi.org/10.25669/uzaq-mrm3>

This Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

DESIGN IMPLEMENTATION AND ANALYSIS OF A DYNAMIC
CRYPTOGRAPHY ALGORITHM WITH APPLICATIONS

by

Sourabh Ghose

Bachelor of Engineering
University of Bombay, India
2005

A thesis submitted in partial fulfillment
of the requirements for the

**Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering**

**Graduate College
University of Nevada, Las Vegas
May 2007**

UMI Number: 1443755

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 1443755

Copyright 2007 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346



Thesis Approval

The Graduate College
University of Nevada, Las Vegas

APRIL 16TH, 2007

The Thesis prepared by

SOURABH GHOSE

Entitled

DESIGN, IMPLEMENTATION AND ANALYSIS OF A DYNAMIC

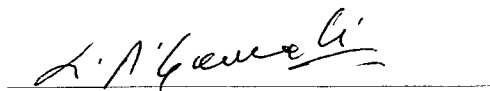
CRYPTOGRAPHY ALGORITHM WITH APPLICATIONS

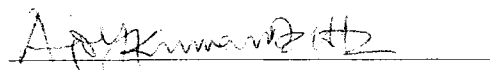
is approved in partial fulfillment of the requirements for the degree of

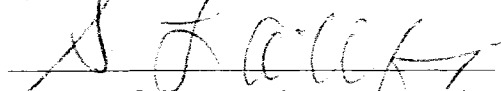
MASTER OF SCIENCE IN COMPUTER SCIENCE


Examination Committee Chair


Dean of the Graduate College


Examination Committee Member


Examination Committee Member


Graduate College Faculty Representative

ABSTRACT

Design, Implementation and Analysis of a Dynamic Cryptography Algorithm with Applications

by

Sourabh Ghose

Dr. Yoohwan Kim, Examination Committee Chair
Assistant Professor of Computer Science Department
University of Nevada, Las Vegas

Cryptographers need to provide the world with a new encryption standard. DES, the major encryption algorithm for the past fifteen years, is nearing the end of its useful life. Its 56-bit key size is vulnerable to a brute-force attack on powerful microprocessors and recent advances in linear cryptanalysis and differential cryptanalysis indicate that DES is vulnerable to other attacks as well. A more recent attack called XSL, proposes a new attack against AES and Serpent. The attack depends much more critically on the complexity of the nonlinear components than on the number of rounds. Ciphers with small S-boxes and simple structures are particularly vulnerable. Serpent has small S-boxes and a simple structure. AES has larger S-boxes, but a very simple algebraic description. If the attack is proven to be correct, cryptographers predict it to break AES with a 2^{80} complexity, over the coming years.

Many of the other unbroken algorithms –Khufu, REDOC II, and IDEA—are protected by patents. RC2 is broken. The U.S. government has declassified the Skipjack algorithm in the Clipper and Capstone chips.

If the world is to have a secure, unpatented, and freely- available encryption algorithm, we need to develop several candidate encryption algorithms now. These algorithms can then be subjected to years of public scrutiny and cryptanalysis. The purpose of the thesis is to discuss the requirements for a standard encryption algorithm.

DCA (**D**ynamic **C**ryptography **A**lgorithm), a new private-key block cipher, is proposed. The block size is user defined, and the key can be of infinite length. The algorithm is a first of its kind dynamic algorithm in which almost all components change depending of the password itself used to generate a key or encrypt a file. The “Key dependency” has been pushed to the extreme with dynamically linked components like number of rounds, operation used on each bits, shift window, direction of each operation (Left or Right), size of the key buffer when encrypting a file, size of the block shuffle and working file block.

The actual encryption of data is performed in two modules, Key generation and File encryption and is very efficient on all microprocessors. Key generation is a complex procedure of Key Padding, Concatenating bits, initial scrambling, key encryption and final scrambling. File encryption consists of complex steps of initialization, seeding and shuffling. All default settings can also be changed by the user, which means the knowledge required to decrypt/reproduce a key gets extended to the environment settings as well as the password itself.

TABLE OF CONTENTS

ABSTRACT.....	iii
LIST OF FIGURES	vi
ACKNOWLEDGEMENTS	vii
CHAPTER 1 INTRODUCTION	1
1.1 Terminology	2
1.2 Algorithm Types	2
1.3 Cryptography Standards	3
1.4 Possible Applications	5
1.5 Common Cryptography Attacks	6
CHAPTER 2 LITERATURE REVIEW	9
2.1 Message Digest Algorithms	9
2.2 Message Authentication code Algorithms	12
2.3 Symmetric Key Algorithms	14
2.4 Block Cipher Algorithms	17
2.5 Asymmetric Key Algorithms	19
CHAPTER 3 ALGORITHM DESCRIPTION	21
3.1 Introduction	21
3.2 Key Generator Overview	23
3.3 Key Generator Details	26
3.4 File Encryption Overview	42
3.5 File Encryption Details	45
CHAPTER 4 ALGORITHM APPLICATIONS	58
CHAPTER 5 ALGORITHM ANALYSIS	61
5.1 Linear Cryptanalysis	61
5.2 Differential Cryptanalysis	66
5.3 Performance Analysis	69
CHAPTER 6 CONCLUSION	75
CHAPTER 7 REFERENCES	78
VITA	87

LIST OF FIGURES

Figure 3.1 Key Padding	26
Figure 3.2 Bits Concatenation	27
Figure 3.3 Initial Scrambling	27
Figure 3.3a Add()-Random Number Generation	28
Figure 3.3b Add() - Pass_code Generation	29
Figure 3.4 Key Encryption	30
Figure 3.4a Swap() – Init	31
Figure 3.4b Swap() - Modulo Generation	32
Figure 3.4c Swap() – Direction	33
Figure 3.4d Swap() – Operation	34
Figure 3.5 Final Scrambling	35
Figure 3.6 Long Key Generation	38
Figure 3.7 Bits Swapping	40
Figure 3.8 Initialization of file encryption	45
Figure 3.8a file_crypt() – Init	45
Figure 3.8b file_crypt() –Seed() - Random Generation	46
Figure 3.9 Seeding	47
Figure 3.10 Shuffling	48
Figure 3.10a file_crypt() – Shuffle() – Initialization	48
Figure 3.10b file_crypt() – Shuffle() - Position & Operation	49
Figure 3.10c file_crypt() – Shuffle() –Last 2 blocks	50
Figure 3.10d Alternative Block Crypt size	51
Figure 3.11 File Encryption and Key Generation	53
Figure 3.12a Plot of Request per second against user load, data = 4 KB	70
Figure 3.12b Plot of Response time against user load, data = 4 KB	71
Figure 3.13a Plot of Request per second against user load, data = 100 KB	72
Figure 3.13b Plot of Response time against user load, data = 100 KB	72
Figure 3.14a Plot of Request per second against user load, data = 500 KB	74
Figure 3.14b Plot of Response time against user load, data = 500 KB	74

ACKNOWLEDGMENTS

I am deeply indebted to my parents for their constant guidance and encouragement. My family has always supported me through my toughest times and it is their belief in me that has brought me this far.

I am very grateful to Dr. Yoohwan Kim, my thesis advisor, teacher and guide for giving me an opportunity to work with him. He has been very patient with me and I owe it to him for all that I have learnt working on this thesis.

I am grateful to Dr. Ajoy Datta, Dr. Laxmi Gewali and Dr. Shahram Latifi for kindly consenting to be a part of my thesis committee.

I have had an excellent work environment right through my two years at UNLV. It has been an enjoyable time working with my peers and the staff of the Computer Science department.

CHAPTER I

INTRODUCTION

Privacy is a sensitive subject that affects everyone. Common techniques to safeguard privacy are:

- When writing a confidential letter an envelope is used to send it.
- When using a credit card a secret code number is used.
- Speaking to someone in private.

In case of computers, sensitive data is common (e.g. assessment grades, financial accounts, etc). With the Internet, the computer can be used like a telephone or like a post office, with the disadvantage that everybody connected to the network could have access to the data. This is why, especially with computers, privacy is important. Different levels of security (computer security, network security, etc) have to be considered.

Cryptography can be compared to an electronic safe where private data is hidden. The cryptographer must always think about the intruder. Cryptography can be compared to a chess game, in that we must think not only of our own tactics, but also of our opponents. Cryptography usually uses a lot of mathematical formulae and logical functions. The science is quite new for the public, this is why it is a very difficult subject, but now more people are interested in it and a lot of books dealing with the subject have been written and it is now easy to find good cryptography information. It seems that the

strongest cryptography algorithms are now available to the public even if it is very difficult to understand them. Indeed, the best way to know if a cryptography algorithm is strong is to make its source code and documentation available to the public. If no one can break it then it is safe to use it.

1.1 Terminology

To make the thesis easier to understand, here are definitions of terms used in the text.

Clear text: An understandable message, usually the original.

Cipher text: An incomprehensible message, usually the result.

Password: A secret string of characters.

Encrypt: Transform a clear text into a cipher text, usually with a password.

Decrypt: Transform a cipher text into a clear text, usually with a password.

Key: Some data that will be used into the message encrypt process. It can also be used like a password, the difference is in this case that it is a long string of characters and numbers one cannot remember as is very long and complex. A key could be compared to a cipher password.

Private Key: This key is personal and only known by one person.

Public Key: This key is available to everybody, it is now secret.

1.2 Algorithms Types

Here is a general overview of these two cryptography standards.

Private Key algorithm:

A private key algorithm uses one password (or one private key) to encrypt a message, to decrypt it the same password is used. The same algorithm or a different one can be used to crypt and decrypt.

Public key algorithm:

A public key algorithm consists of a public key (B) used to encrypt a message and a private key (A) used to decrypt the message, for one public key there is one private key (A1, B1) and only the private key that belongs to the public key can decrypt a message encrypted by the public key.

Due to this, everyone can use the public key, if someone wants to send message, they encrypt the message with the receiver's public key, and only the receiver, who knows the private key, can decrypt this message.

This algorithm can also be used to sign a message to prove that it is really the sender who is sending a message, to do so the sender encrypts the message with his private key that can be decrypted only with the public key.

1.3 Cryptography Standards

Introduction

A lot of cryptography algorithms have been created, it is not the aim of this report to go into great detail about cryptography, so only two of the most famous and used cryptography algorithms are going to be quickly explained to give a general idea of how to encrypt a message.

DES

DES stands for "Data Encryption Standard" and is at the moment the most used algorithm in the world, being used by the American government to secure their sensitive data. It has been created by IBM (International Business Machines Corporation) in 1977 and is a private key algorithm.

It is a block cipher algorithm that crypts data by 64 bits length block, that means that the clear text is divided into 64 bits length block and each block is encrypted by 16 complex operations. The entire 64-bit length encrypted block constitutes the final cipher text. The decryption algorithm is nearly the same as the encryption algorithm, the same key (the private key) is used to encrypt and to decrypt a message; the bigger the private key is the safer it is.

RSA

RSA is the initial of the name of its creator: Ron RIVEST, Adi SHAMIR and Leonard ADLEMAN. It is one of the first public key algorithms and was created in 1978. In fact there are two algorithms, one to generate the keys and one to encrypt/decrypt the message; the pair of keys, one public and one private, are based on big first numbers and are the result of some calculations (modulo, Euclid's algorithm, etc). The algorithm that encrypts/decrypts a message is a block cipher algorithm that is simpler than the DES algorithm but is much slower.

The security of this algorithm is based on mathematical theories (big numbers factorization), even if no real proof has been given to demonstrate that these mathematical theories are not easily "breakable", they have not been broken for 20 years.

These two algorithms have different concepts, but neither of them is better than the other, they have their own advantages. A good idea is to use both of them choosing

which one depending on its suitability to a specific job. It is why DES is usually used to encrypt the message, and RSA is used to communicate only the DES private key used to encrypt the message.

The reason for this choice is because DES is faster than RSA and more difficult to break. However, to communicate the DES private key a secure solution has to be found. RSA has a good level of security and the user does not have to send his private key to the recipient, only the public key is to be transmitted. This is why RSA is used to encrypt the DES private key, followed by the message and then sent to the recipient.

1.4 Possible Applications

The aim of this section is to give concrete examples of professional cryptography used, which will help the reader to appreciate more the work done on the project.

1] Login password: Here the password typed by the user is encrypted and compared to with the user's encrypted password stored in the password database. If the user did not make any mistakes while typing his password then the two cipher texts will be identical and the user is allowed to log onto the system.

2] Pretty good Privacy: This is a famous application for encrypting personal data such as letters, emails, a file or anything else found on a computer; because it is very powerful and has been developed on almost all existing computers (PC, Macintosh, Amiga, etc). In fact, it is the first application that has been developed for public as a pose to military use. It is maybe why PGP is becoming a standard application on computers. The only problem is the lack of a good Graphical User Interface (GUI).

1.5 Common Cryptography Attacks

Introduction

There are different ways to attack a cryptography algorithm. If the algorithm security is only based on its secret, once someone finds the algorithm source code it will be very easy to break it.

Algorithm attacks

Sometimes a cryptography algorithm can have weak points, for example, with algorithms that just consist of adding the same number to all the password letters they are easier to break. This is because it is simple; all that is to be done is to find the number used. This attack needs strong cryptography knowledge and understanding. It is only used for bad cryptography algorithms, but as with everything relating to computer science; it is very difficult to totally avoid making errors, so this attack is always the first one attempted. If no weak points are found the only attack that can be done is the "brute force" attack. Some advanced techniques used are Linear and Differential cryptanalysis which will be discussed later in the thesis.

Brute force attack

This attack is based on the cipher text generated by a cryptography algorithm. If the attacker can get into the password database, and even if all the passwords in it are encrypted, software exists that simply try every possible passwords, encrypts them and compares the cipher text generated with the one held in the password database. Of course, it would take too long to try all the possible passwords, present computers are not fast enough and there are too many possibilities.

Keeping these in mind, Cryptographers need to provide the world with a new encryption standard. Many of the unbroken are protected by patents. If the world is to have a secure, unpatented, and freely- available encryption algorithm, we need to develop several candidate encryption algorithms now. These algorithms can then be subjected to years of public scrutiny and cryptanalysis. Then, the hope is that one or more candidate algorithms will survive this process, and can eventually become a new standard. The purpose of the thesis is to discuss the requirements for a standard encryption algorithm. While it may not be possible to satisfy all requirements with a single algorithm, it may be possible to satisfy them with a family of algorithms based on the same cryptographic principles. It introduces a new cryptographic algorithm and goes on to discuss how the algorithm overcomes known attacks like Linear and Differential cryptanalysis. It identifies a potential weakness in the algorithm and goes on to describe how the algorithm overcomes it.

DCA (Dynamic Cryptography Algorithm), a new private-key block cipher, is proposed. The block size is user defined, and the key can be of infinite length. The algorithm is a first of its kind dynamic algorithm in which almost all components change depending of the password itself used to generate a key or encrypt a file. The “Key dependency” has been pushed to the extreme with dynamically linked components like number of rounds, operation used on each bits, shift window, direction of each operation (Left or Right), size of the key buffer when encrypting a file, size of the block shuffle and working file block.

The actual encryption of data is performed in two modules, Key generation and File encryption and is very efficient on all microprocessors. Key generation requires a

seed as input, whose size is $N/2$ for key size N . Key generation is a complex procedure of Key Padding, Concatenating bits, initial scrambling, key encryption and final scrambling. File encryption consists of complex steps of initialization, seeding and shuffling. All default settings can also be changed by the user, which means the knowledge required to decrypt/reproduce a key gets extended to the environment settings as well as the password itself.

CHAPTER II

LITERATURE REVIEW

Several Cryptography algorithms have been designed in the past. Discussing all of them is beyond the scope of this project. However the major ones have been listed according to their type.

2.1 Message Digest Algorithms

As an Internet standard (RFC 1321), message digest algorithms have been employed in a wide variety of security applications, and is also commonly used to check the integrity of files. A message digest hash is typically a 32-character hexadecimal number. Recently, a number of projects have created message digest "rainbow tables" which are easily accessible online, and can be used to reverse many hashes into strings that collide with the original input.

MD5:

Designer:

Ron Rivest

References:

[16], [17], [18], [19], [20]

Digest length:

16 bytes.

Block size:

64 bytes.

Max. final block size:

55 bytes.

State size:

16 bytes.

Comments:

- A transcription error was found in the original MD5 draft RFC. The corrected algorithm should be called MD5a, though some people refer to it as MD5.

This is wrong; the corrected algorithm should be called MD5, and is in practice never referred to as MD5a.

- MD5 is big-bit-endian, little-byte-endian, and left-justified.

Security comment:

Hans Dobbertin has found a method of generating collisions for MD5's compression function. Quoting from RSA Laboratories Security Bulletin #4:

“Given the surprising speed with which techniques on MD4 were extended to MD5 we feel that it is only prudent to draw a cautious conclusion and to expect that collisions for the entire hash function might soon be found. In addition, the 128-bit output is arguably not long enough to make generating collisions using a birthday attack infeasible.”

Ripemed-320:

Designers:

Hans Dobbertin, Antoon Bosselaers, Bart Preneel

Published:

April 1996

Alias:

"RIPEMD320"

References:

[21], [22], [23], [24]

Digest length:

40 bytes.

Block size:

64 bytes.

Max. final block size:

55 bytes.

State size:

20 bytes.

Comment:

RIPEMD-320 is big-bit-endian, little-byte-endian, and left-justified.

Security comment:

This message digest is not claimed to provide a security level higher than RIPEMD-160. SHA-384, SHA-512 or Whirlpool is used instead.

SHA-512:

Designers:

U.S. National Security Agency

Published:

October 2000

References:

[25]

Digest length:

64 bytes.

Block size:

128 bytes.

Max. final block size:

111 bytes.

State size:

64 bytes.

Comment:

- SHA-{256,384,512} are big-bit-endian, big-byte-endian, and left-justified.
- When the compression function is used directly, it is considered to include the chaining variable addition (as opposed to being separate as shown in the specification).

2.2 Message Authentication Code Algorithms

A cryptographic message authentication code (MAC) is a short piece of information used to authenticate a message. A MAC algorithm accepts as input a secret key and an arbitrary-length message to be authenticated, and outputs a MAC (sometimes known as a tag). The MAC value protects both a message's integrity as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content. A Message Integrity Code (MIC) is another name for a MAC.

HMAC (Digest):

Designers:

Mihir Bellare, Ran Canetti, Hugo Krawczyk, Adi Shamir

Published:

June 1996

Aliases:

- "HmacMD5" is an alias to HMAC(MD5) [JDK compatibility]
- "HmacSHA1" is an alias to HMAC(SHA-1) [JDK compatibility]
- "1.3.6.1.5.5.8.1.1" is an alias to HMAC(MD5)
- "1.3.6.1.5.5.8.1.2" is an alias to HMAC(SHA-1)
- "1.3.6.1.5.5.8.1.3" is an alias to HMAC(Tiger)
- "1.3.6.1.5.5.8.1.4" is an alias to HMAC(RIPEMD-160)

(source for OIDs from

iso.org.dod.internet.security.mechanisms.ipsec.isakmpOakley tree)

- "http://www.w3.org/2000/02/xmldsig#hmac-sha1" is an alias to HMAC(SHA-1)

References:

[26], [27], [28], [29], [30]

Parameters:

- String digest [creation/read, no default] - the name of the Block MessageDigest on which this MAC is to be based.

Key length:

Any multiple of 8 bits that does not cause the maximum input length for the MessageDigest to be exceeded. Default 128 bits.

Output length:

Minimum 32 bits, maximum equal to the message digest output length. The default is equal to the message digest output length.

2.3 Symmetric Key Algorithms

Symmetric-key algorithms are a class of algorithms for cryptography that use trivially related cryptographic keys for both decryption and encryption.

The encryption key is trivially related to the decryption key, in that they may be identical or there is a simple transform to go between the two keys. The keys, in practice, represent a shared secret between two or more parties that can be used to maintain a private information link.

Other terms for symmetric-key encryption are single-key, one-key and private-key encryption. Use of the latter term can sometimes conflict with the term *private key* in public key cryptography.

AES256 (Advanced Encryption Standard):

Designers:

Joan Daemen, Vincent Rijmen

Alias:

"OpenPGP.Cipher.9"

Description:

AES256 is defined as Rijndael with a 128-bit block size and 14 rounds.

References:

[31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41]

Key length:

256 bits.

Block size:

16 bytes.

Security Comment:

- There are claims that the XSL (Extended Sparse Linearization) [42] attack can break the AES. Since AES is already widely used in commerce and government for the transmission of secret information, finding a technique that can shorten the amount of time it takes to retrieve the secret message without having the key would have wide implications. Opinions differ on whether the attack works because the method is heuristic and very technical, and so it has proved difficult to evaluate its complexity. In addition, the method is expected to have a high work-factor, which unless lessened, means the technique would not reduce the effort to break AES very much in comparison to an exhaustive search. Therefore, even if the attack has been analyzed correctly, it is unlikely to affect

the real-world security of block ciphers in the near future. Nonetheless, the attack has caused some experts to express greater unease at the algebraic simplicity of the current AES.

DES (Data Encryption Standard):

Designers:

Don Coppersmith, Horst Feistel, Walt Tuchmann, U.S. National Security Agency

Published:

1976

References:

[43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56]

Key length:

64 bits as encoded; 56 bits excluding parity bits.

Block size:

8 bytes.

Comment:

Implementations MUST ignore (i.e. not check) the parity bits of keys.

KeyGenerators for DES MUST, however, output keys with correct parity.

Security comment:

The fixed 56-bit effective key length is too short to prevent brute-force attacks [2]

DES is also vulnerable to Differential and linear cryptanalysis.

RC4:

Designer:

Ron Rivest

Published:

September 1994

Alias:

"ARCFOUR"

References:

[57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71],
[72], [73]

Key length:

Minimum 8, maximum 2048, multiple of 8 bits; default 128 bits.

Security comments:

- There are small biases at the start of the RC4 keystream.
- The RC4 keystream is distinguishable from random given about 2 Gbytes of the stream.
- RC4 is vulnerable to related-key attacks, and therefore it should only be used with keys that are generated by a strong RNG, or by a source of bits that are sufficiently uncorrelated (such as the output of a hash function).

2.4 Block Cipher algorithms

In cryptography, a block cipher is a symmetric key cipher which operates on fixed-length groups of bits, termed blocks, with an unvarying transformation. When encrypting, a block cipher might take a (for example) 128-bit block of plaintext as input, and output a corresponding 128-bit block of cipher text. The exact transformation is controlled using a second input — the secret key. Decryption is similar: the decryption

algorithm takes, in this example, a 128-bit block of cipher text together with the secret key, and yields the original 128-bit block of plaintext.

ECB (Electronic Codebook):

Description:

Electronic Codebook Mode, as defined in NBS FIPS PUB 81.

References:

[74], [75]

Comment:

If a padding scheme is not specified (i.e. the algorithm name is given in the form "*cipherName*/ECB"), then NoPadding MUST be assumed (note that this is intentionally different to CBC and PCBC modes, for which PKCSPadding would be used). The standard name always specifies which padding method is used, i.e. it always has three components.

Security comment:

ECB mode will always encrypt identical plaintext blocks to identical ciphertexts. This can be a weakness when the plaintext is not random, uniformly distributed, and a multiple of the block size. If these conditions are not satisfied, a different mode should probably be used.

CBC (Cipher Block Chaining):

Description:

Cipher Block Chaining Mode, as defined in NBS FIPS PUB 81.

References:

[76], [77], [78]

Comment:

If a padding scheme is not specified (i.e. the algorithm name is given in the form "*cipherName/CBC*"), then PKCSPadding_MUST be assumed. The standard name always specifies which padding method is used, i.e. it always has three components.

2.5 Asymmetric Key algorithms

RSA:

Designers:

Ron Rivest, Adi Shamir, Leonard Adelman

Aliases:

"RSAES", "1.2.840.113549.1.1.1", "2.5.8.1.1"

References:

[79], [80], [81], [82], [83], [84]

Comment:

It is recommended that implementations make no practical restriction on the lengths of the key parameters n and e (in particular, values of n up to at least 4096 bits SHOULD be supported).

Security Comments:

- The most damaging would be for an attacker to discover the private key corresponding to a given public key; this would enable the attacker both to read all messages encrypted with the public key and to forge signatures. The obvious way to do this attack is to factor the public modulus, n , into its two

prime factors, p and q . From p , q , and e , the public exponent, the attacker can easily get d , the private exponent. The hard part is factoring n ; the security of RSA depends on factoring being difficult. In fact, the task of recovering the private key is equivalent to the task of factoring the modulus: use d to factor n , as well as use the factorization of n to find d .

Patent status:

RSA is patented in the United States and Canada (see references); the patent is licensed by RSA Data Security, Inc.

CHAPTER III

ALGORITHM DESCRIPTION

3.1 Introduction

The Dynamic Cryptography algorithm has the following features:

- Private Key Algorithm
- Dynamic Cryptography Algorithm
- Source code can be public without making the algorithm weak
- Multiplatform application
- Infinite key length (as big as the integer type being used)
- Bilateral bits swapping with variable windows
- Bilateral Pseudo randomly binary operations
- Dynamic Variables changing in functions of the password, such as:
 - Round, Block Shuffle, etc
 - Key buffering against key dependency attacks
 - Addition of a random number to the key
 - Random Number Generator (RNG) using the ISAAC Algorithm
 - Possibility to specify own RNG seed
- 5 different crypt's level (allowing to choose between efficiency and speed)
- Seed and shuffle functions

- Seed and shuffle functions
- A clear text can be encrypted using its own data
- Two methods of execution: direct disk access or memory cache
- Strong Key generator

The DCA has been made as dynamic as possible; this means almost all of its components will change depending of the password itself used to generate a key or encrypt a file. This is called “Key Dependency” (**KD**) and has been pushed to the extreme in the DCA with dynamically linked components such as:

- The number of rounds
- The operation used on each bits
- The shift window
- The direction of each operation (Left or Right)
- The size of the key buffer when encrypting a file
- The size of the block shuffle and working file block.

The algorithm behaviour changes to great extent when used with different passwords.

All default settings can also be changed by the user, which means the knowledge required to decrypt/reproduce a key gets extended to the environment settings as well as the password itself.

Algorithm Overview:

The algorithm is made of a number of modules and sub-modules.

- The Key Generator module
- The File Encryption module

The “file decryption” module has been left out in purpose as it is nothing more than running the “file encryption” module backwards.

3.2 Key Generator Overview

Pre-requisites

There are a number of pre-requisites to this module:

- Seed source:

The Seed can be a password, a key, or the result of a random algorithm (*ISAAC*, time/date based, or user customised).

- Keylength Output:

The minimum keylength that can be generated is 128bits. Larger keys will be a multiple of 128. There is no upper limit to the size of key.

- Keylength Input:

The seed must be at least of size $N/2$ when generating a key of size N . In other words, when generating a 128bits key and using a password, the user must enter a password of at least 8 characters ($8 \times 8\text{bits} = 64\text{bits}$).

Terminology

KD – Key Dependant – a value which is “Key Dependant” will be different each time a different key is used.

PRN – Pseudo Random Number – In general, all references to Random Number in this document should be taken as a reference to a Pseudo Random Number.

LO – Logical Operation – This can be also referenced as a **XOR, AND, NOR, NAND**, etc

LFSR - Linear Feedback Shift Register – Algorithm used to generate **PRN** with the least repetition possible for its output numbers.

Integer Size is 32 bits – The use of a 32 bits hardware platform is assumed.

ISAAC – This is a fast cryptography random generator created by Bob Jenkins and used in the **DCA**. The details of the **ISAAC** Algorithm will not be discussed in this document.

Below is a description of the different Key Generator steps:

- STEP 1: Key Padding

If the initial seed used to generate the key is not equal to the size of the keylength to be generated some Pseudo-Random numbers (**KD**) will be inserted at a position with is **KD**.

- STEP 2: Bits Concatenation

The seeds bits will be stored into one long string of bits. An Integer array will be used for this purpose and the size of each element will be dependant of the hardware platform used: 64, 32 or 16 bits. (or even 128, 256, etc when available).

- STEP 3: Initial Scrambling

The different seed bits will be combined together to generate a Pseudo Random Number (which is therefore **KD**). This PRN will then be added to the each seed element. Each time the PRN is added it will change (**KD**).

- **STEP 4: Key Encryption**

The seeds will now be referenced as the key. Each of its bits will be treated individually and will be subject to some Logical Operations (**LO**). This is called “a round”. In each round the following happens:

- A bit swap or a **LO**
- The distance between 2 bits (Shift Window) is **KD**
- The nature of the operation (a swap or **LO**) is **KD**
- The number of round is **KD**
- The direction of the round is **KD** (left or right)

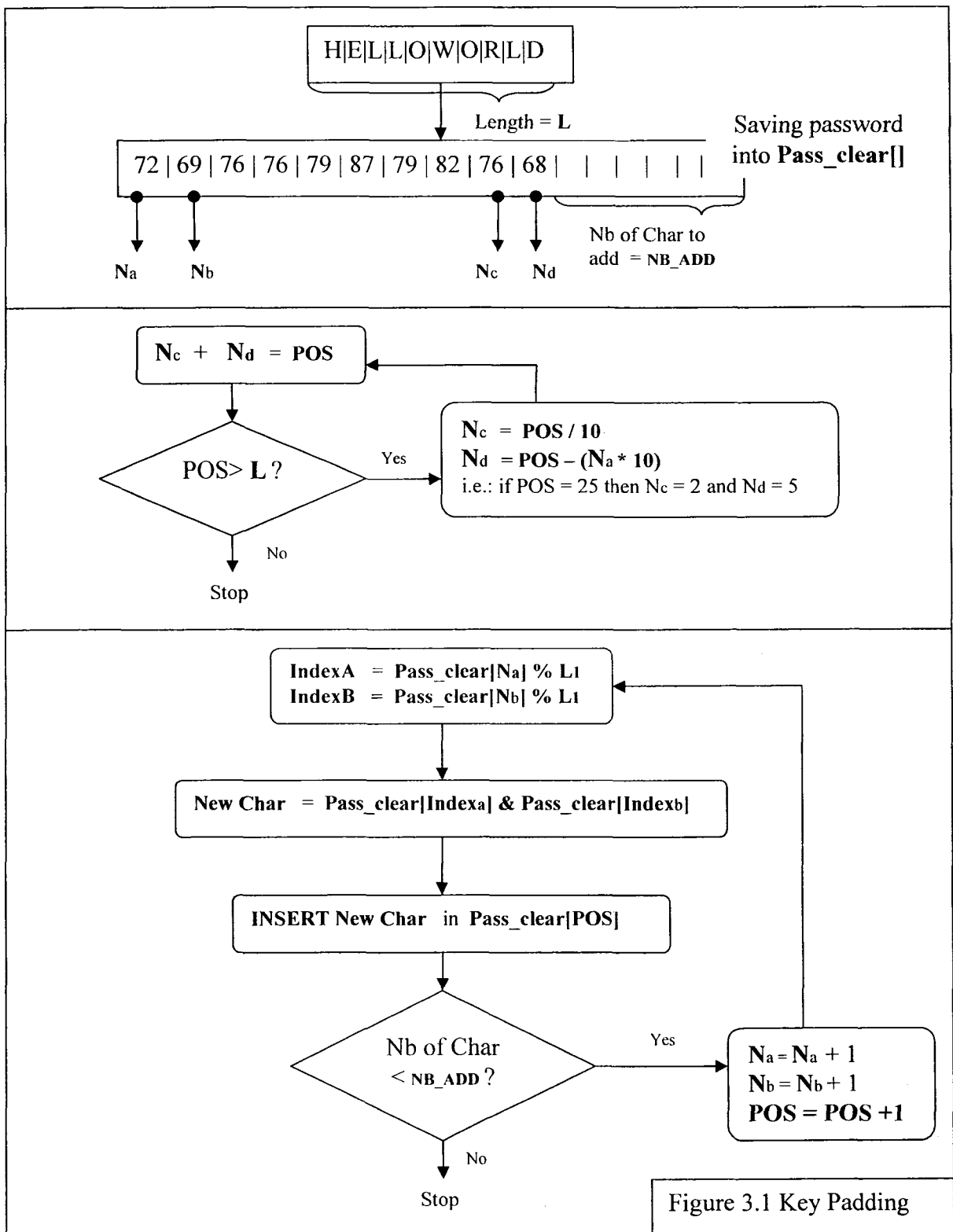
For each key generated the minimum number of rounds is two, this will ensure that all bits have been swapped at least once AND have had a **LO**.

The number of rounds is also **KD**.

- **STEP 5: Final Scrambling**

A PRN is generated if no random seed is provided and will be added to each element of the Key. Each time the PRN is added it is changed using a Linear Feedback Shift Register (**LFSR**)

3.3 Key Generator Details



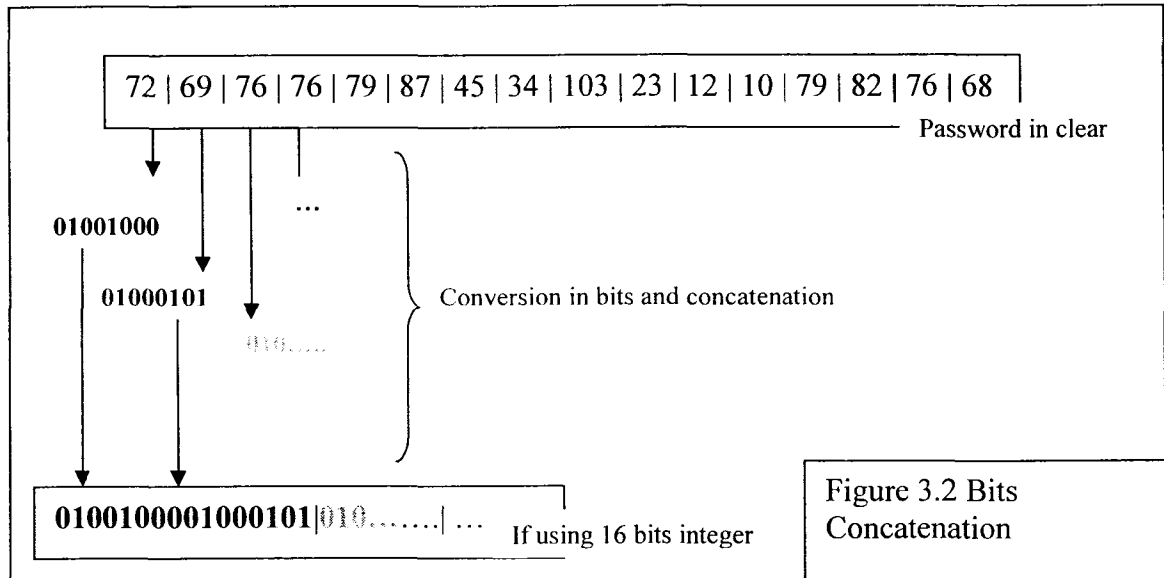
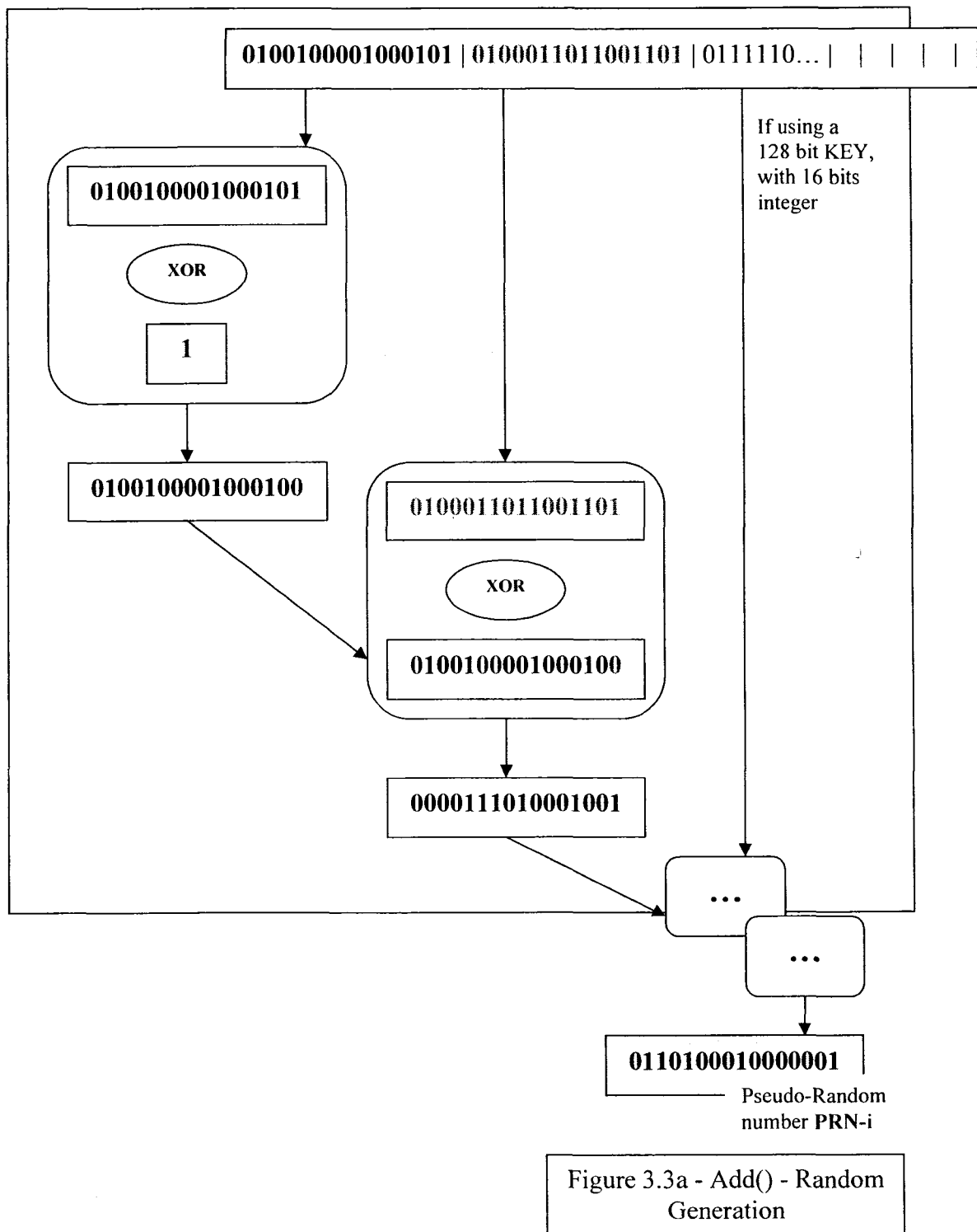


Figure 3.3 Initial Scrambling

See Figure 3.3a - Add() - Random Generation

and Figure 3.3b - Add() - Pass_code Generation



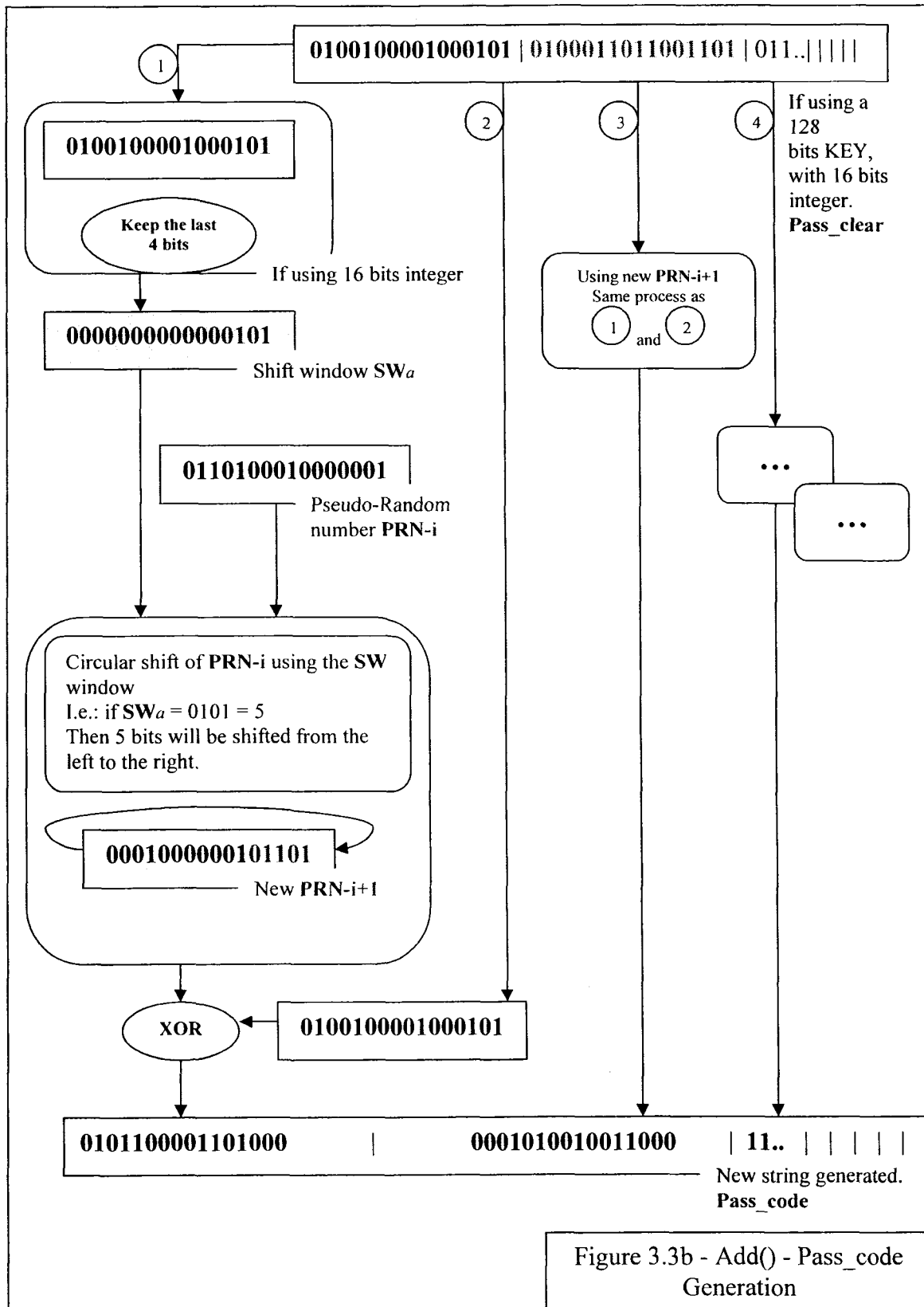
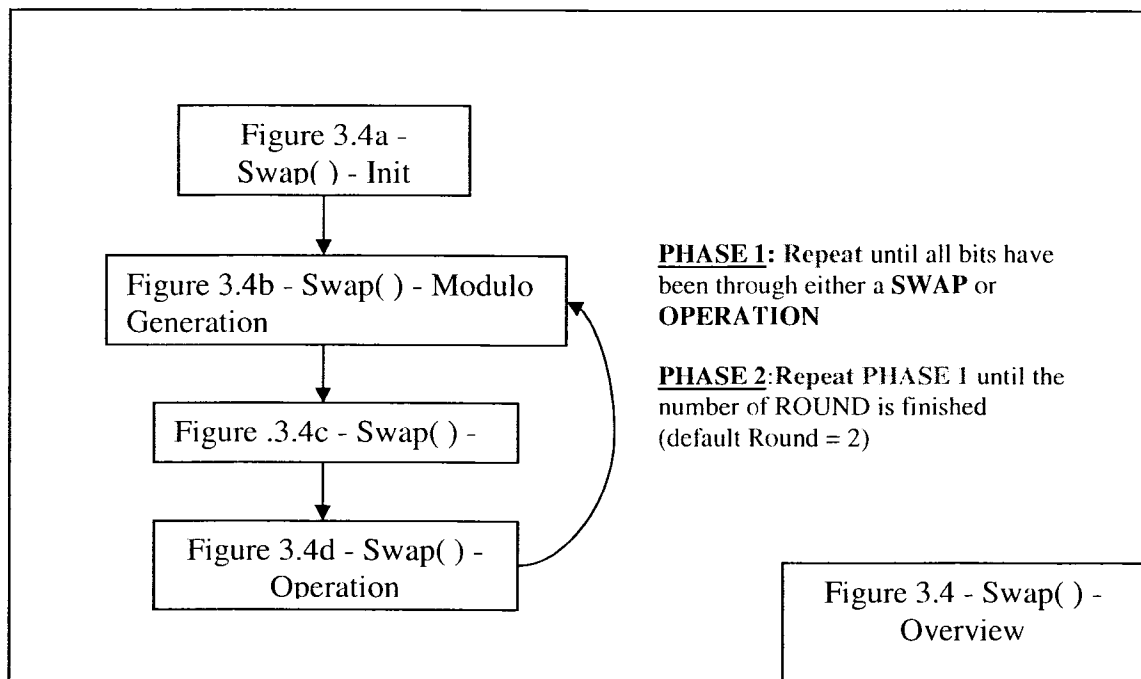


Figure 3.3b - Add() - Pass_code Generation

The following diagram describes an overview of this process:



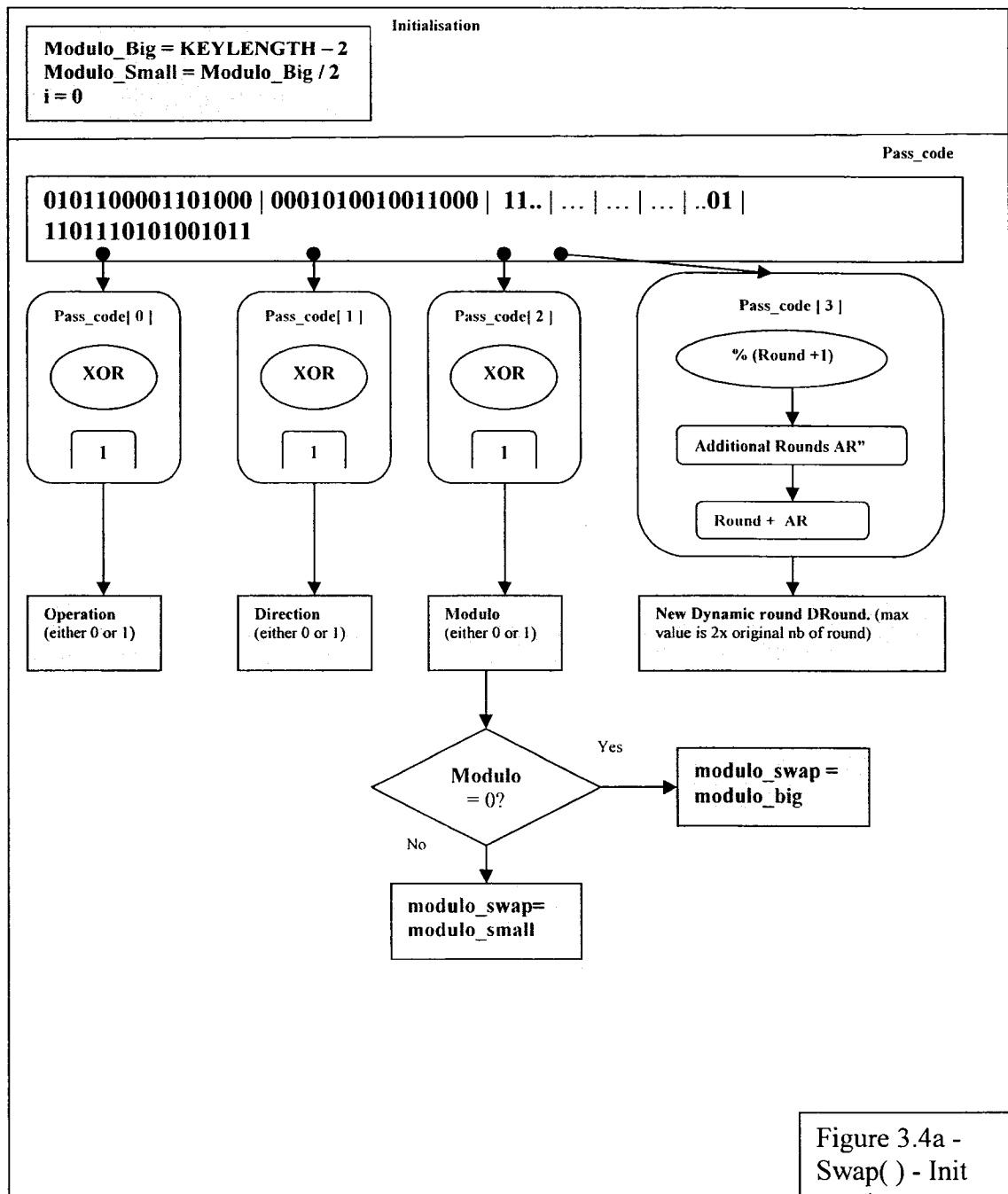
The following 4 diagrams describe the detailed process:

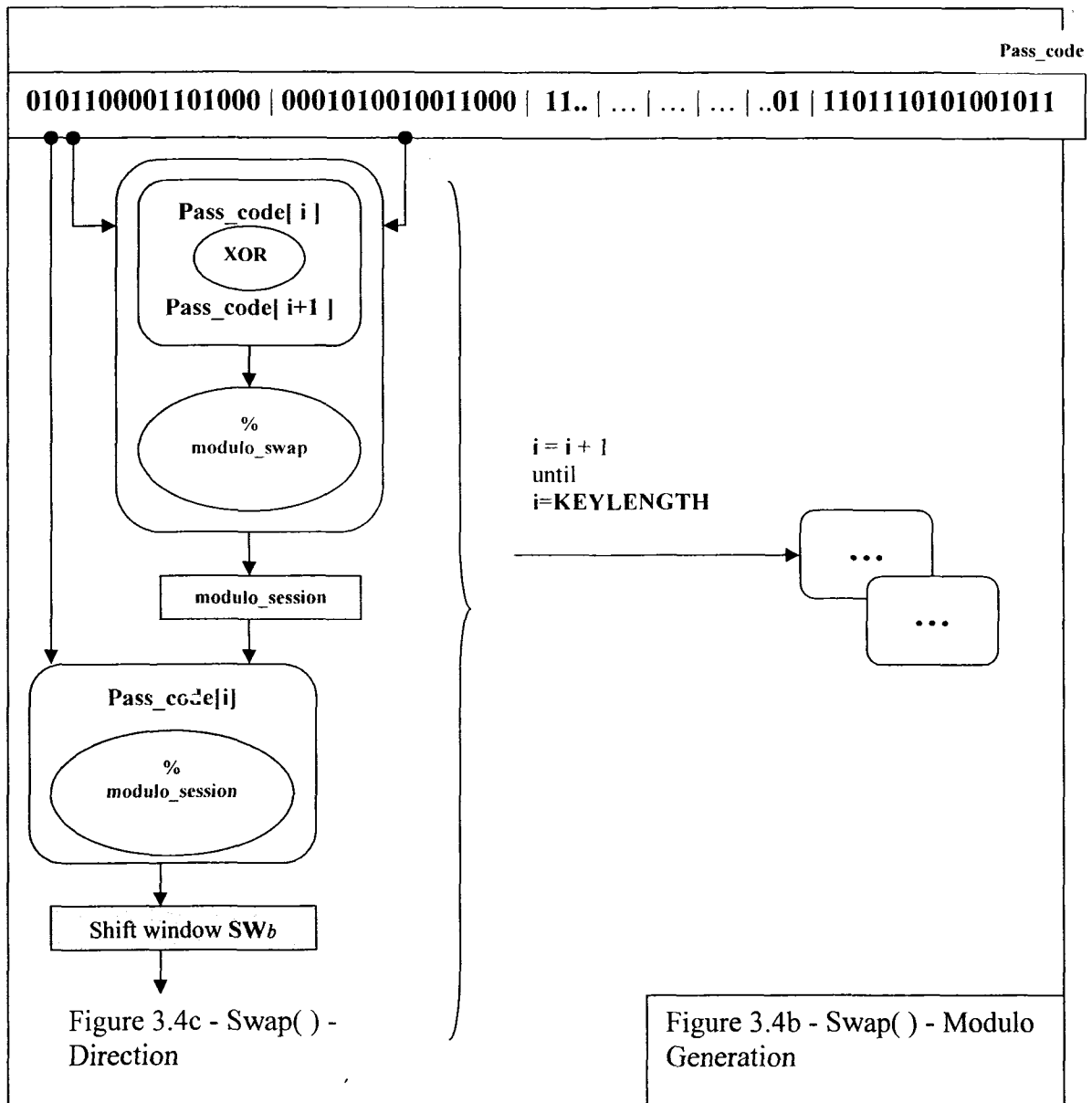
Figure 3.4a - Swap() - Init

Figure 3.4b - Swap() - Modulo Generation

Figure 3.4c - Swap() - Direction

Figure 3.4d - Swap() - Operation





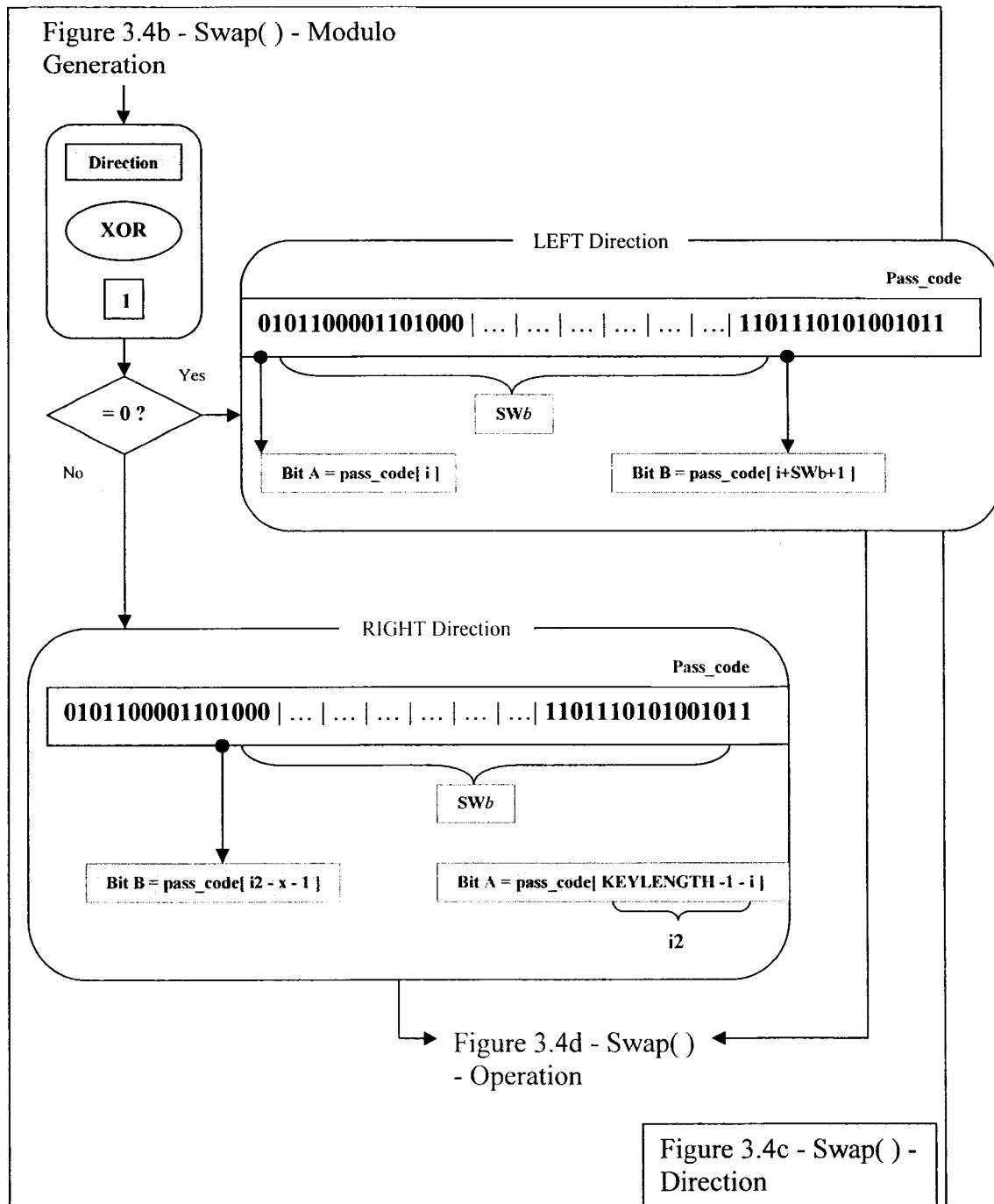
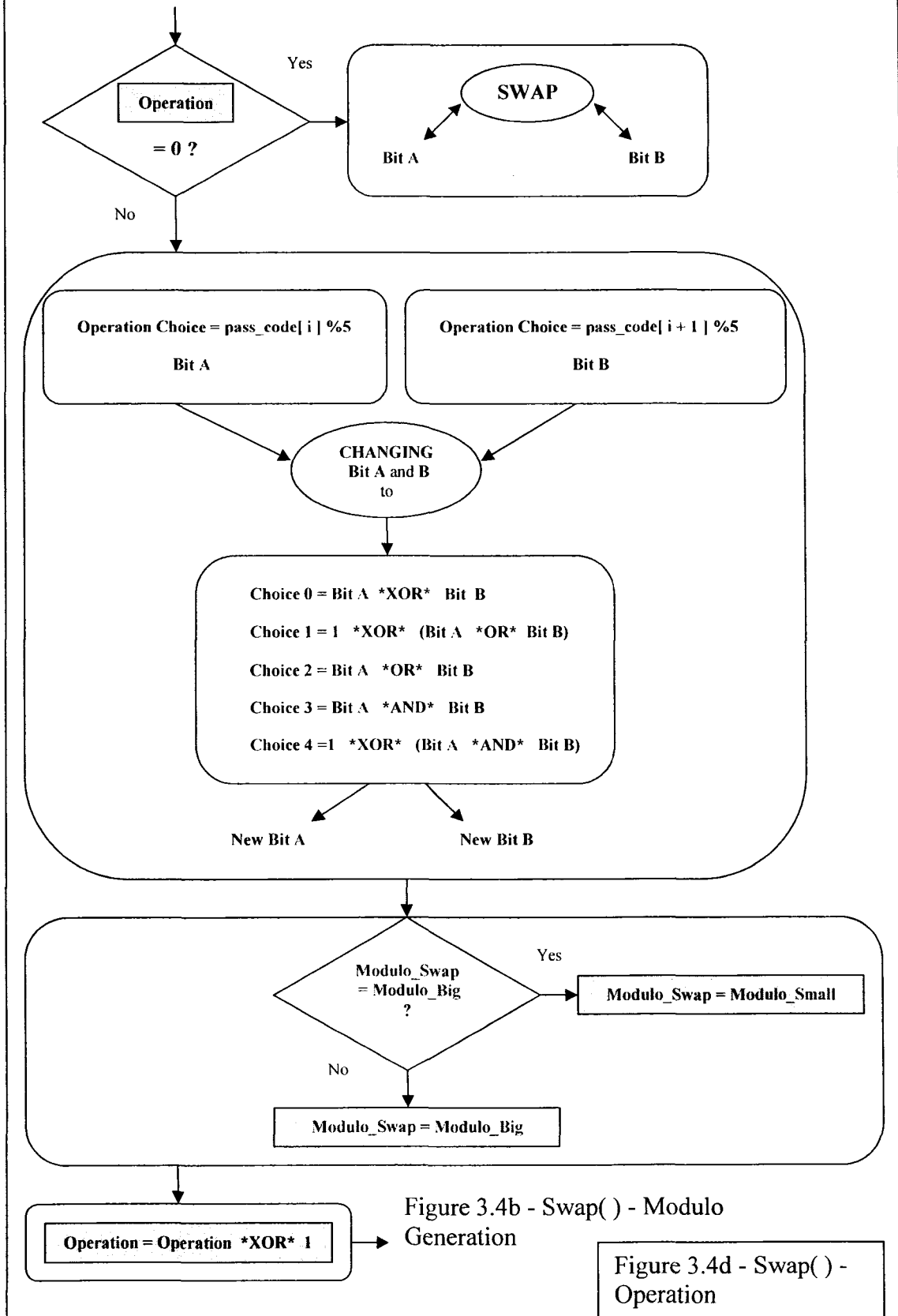
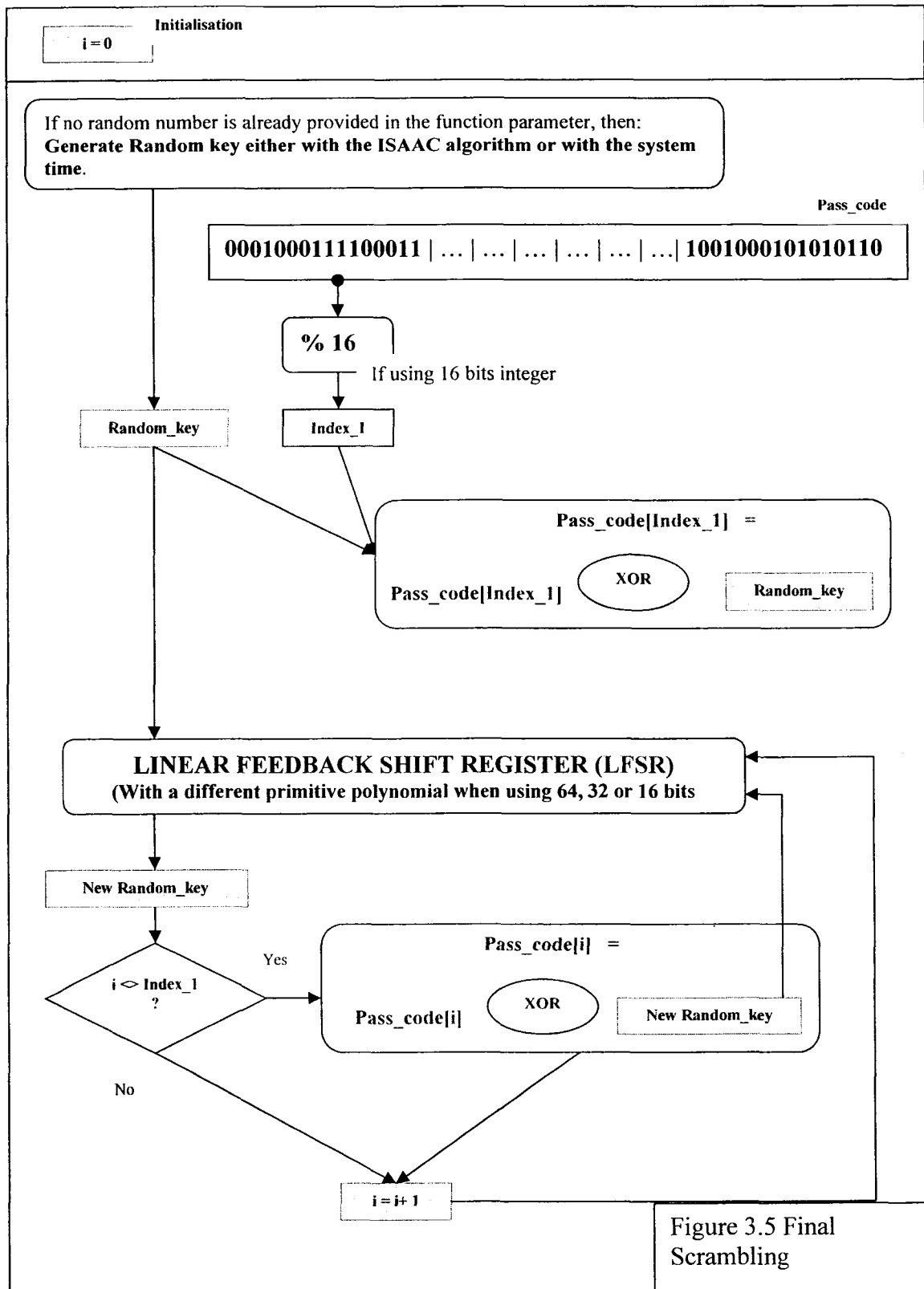


Figure 3.4c - Swap() - Direction





The important functions are discussed below with an example for 512 bits key generation.

Key generator

The key lengths that are generated are powers of two (e.g. 2^x , 128 bits, 256, 512, 1024, etc). To generate a key there are two different methods: randomly or pseudo-randomly. For the random method, 16 random characters are generated and stored in the memory. For the pseudo-random method, the user is asked to type some characters that are then stored in the memory, the maximum number of characters that can be entered is 16. 16 characters correspond to a key length of 128 bits, this is because one character is represented by 8 bits on a computer, therefore 16 characters represent 128 bits ($16 * 8 = 128$), 32 characters represent 256 bits ($32 * 8 = 256$) and so on. Then several functions are used to generate a key, using these characters which are stored in the memory like a password, this string of characters is called: "pass_clear".

The number of different combinations is higher with the Random method (2^{128}) than with the Pseudo-Random method (72^{16} i.e. approximately 2^{99}). This is because the user has to type some characters and there are only around 72 symbols readily available on a standard keyboard. If only un-shifted alphanumeric characters are used then the round falls to around 2^{83} combinations.

Test length function

It is important that the key's length is always the same, so if the length of the "pass_clear" string is inferior to the key's length, some characters are then added to the string. The number of characters the user has to type must be at least half the number of characters in the key. Each new added character is the result of an AND logical operation between two characters randomly picked up in the "pass_clear" string of

characters. Now, if the key to be generated is 128 bits (which correspond to 16 characters) the new "pass_clear" length is 16 characters.

When generating a key of more than 128 bits, first a 128 bit key length is generated and used to generate a 256 bit key length, and so on. This is possible because the first key has got the minimum number of characters required to generate the following key (128 bits = 16 characters, 256 bits = 32 characters, 512 bits = 64 characters, etc). The key generated then becomes the password for the following key. The different steps during a 512 bits key length generation are:

- The user types 16 characters as a password.
- If less than 16 characters are typed; some new characters are generated to reach the 16 characters needed.
- A key (A1) of 128 bits is generated from the password typed at the step 1.
- The key (A1) is used to generate another key.
- The key (A2) of 256 bits is then generated from the Key (A1).
- Because the key (A1) is twice inferior to the key (A2), the same process used at the step 2 has to be done, but this time, we check if there are at least 32 characters (256 bits) in (A1).
- The key (A2) is used to generate another key.
- We repeat step 6 but this time with (A2).
- The final key (A3) is a 512 key length.

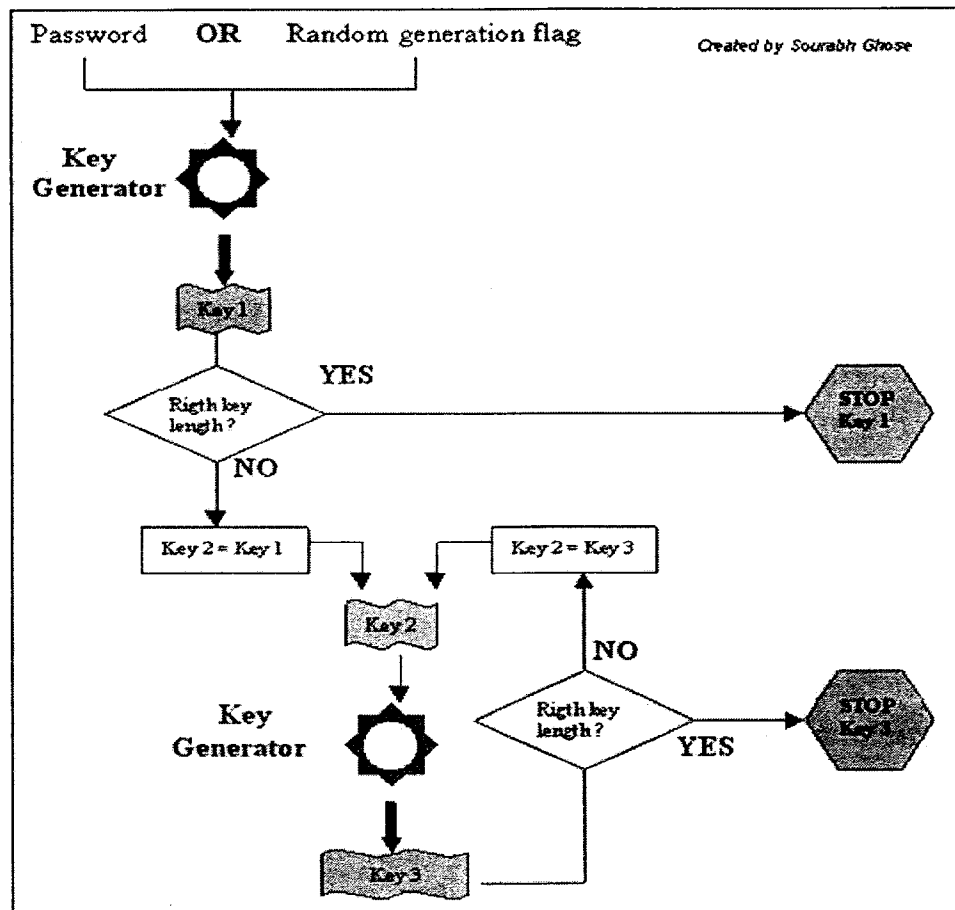


Figure 3.6 Long key generation

Transcription function

The string of character "pass_clear" is converted into a string of numbers called "pass_code", these numbers are the ASCII code of each character contained in the "pass_clear" string.

Additional function

It is better not to directly manipulate the ASCII code contained in the "pass_code" string. This is why a number will be added to each number contained in this string by an Exclusive OR operation. This number is generated in function of the password itself. To

make it safer, each time the number is added it is different we do this by using a circular bit shifting.

Swapping function

This is the main part of the algorithm, all the numbers contained in the `pass_code` string are considered to be only one long string of bits. First, the bits are swapped with each other; each swap takes place between two bits that have a distance of X bits.

The cipher password is a long string of integer, each integer of this string will be assigned to X . Because while doing this bit swapping process the cipher password will be consistently changing so will the string of integer and therefore the value of X . X should never have the same value. The swap starts from the bit B_1 at the position 0 with the bit at the position $B_1 + X_1$. After, from the bit B_2 at the end of the string with the bit at the position $B_2 - X_2$. This continues with the second bit of the string B_3 and the bit $B_3 + X_3$, and with the next to last bit B_4 with the bit $B_4 - X_4$, and so on. In fact, as shown in, the following rule is respected:

Swap bit I with bit $I + X_1$

Swap bit `'endofstring' - I` with bit `'endofstring' - X_2`

This starts again with $I = I + 1$ and a different X from the `"swap_length"` array.

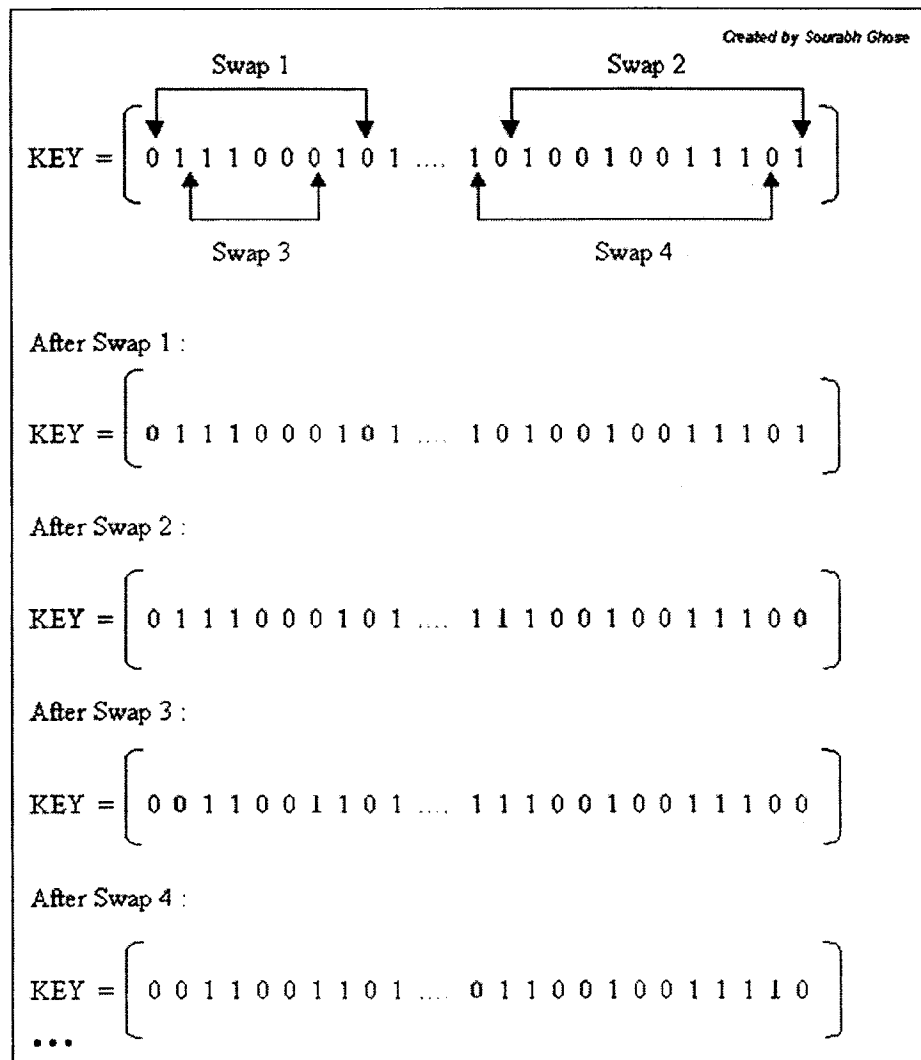


Figure 3.7 Bit swapping

This is called bilateral bit swapping, with this function all the bits will be swapped around.

The second part of this swapping function is the same process, but this time instead of swapping bits around, a pseudo-random binary operation, function of the password itself, will be generated. There are 5 different types of binary operations: Exclusive OR, NAND, NOR, OR, AND.

The way the algorithm works is that it alternates a swap and a logical operation each time. The algorithm does this in a loop until all the bits have been used (in a swap or a logical operation), the user can change the "round" of this loop with a parameter: it is the number of loops the user wants the algorithm to perform. By default it is 2 (which is also the minimum) as with this value we are sure that ALL the bits have been used in a swap AND a logical operation.

By changing the value of the round the user is changing the result generated.

Also, the algorithm will:

- Start from the right or the left of the bits string
- Start by a swap or a logical operation.

When this is finished there is a new "pass_code" string, which contains numbers totally different from the start.

Coding function

To make this encryption algorithm more efficient, a final part has been created. A random number is added to each number of the "pass_code" string. This random number is initialized at the beginning of the function. Each time this random number is added, its bits are shifted to the left, the value of this shift is a number generated in function of the character contained in the "pass_clear" string. Therefore, each time the random number is added, it is different. Due to the random number, even if the user types the same password to generate the key, he can have 2^{31} different keys (if we use a long random number = 4 bytes = 32 bits).

The random number will be hidden in the key at a position that will depend on the cipher string, because when a user types his password again we need to recover it. The

key generation will then follow the same process as the first time it was done. To recover this random number, if the password is right the algorithm will know its position. Otherwise, the position will be wrong as will the random number used in the coding function and the key generated will be different.

The user can choose 2 ways to generate a random number: using the standard C function or use the ISAAC algorithm created by Bob Jenkins. The default RNG is ISAAC. The initial seed will be initialized by the /dev/random or /dev/urandom device if present on the system, otherwise it would be the result of “time() + clock()”. We are aware that the second method hasn’t got a big entropy for the pool of numbers, therefore the user can overwrite the seed value in his application really easily (for example: after the binit() call just add the line: varinit->SEED = 666, to set the SEED to 666).

3.4 File Encryption Overview

Process

Below is a brief description of the different File Encryption steps

- STEP 1: Initialization
 - o The file is mapped into a virtual array and spit into blocks. The length of the block is **KD**.
 - o Several keys are generated from the password or keyfile
 - o Only one key will be generated with a random number, encrypted and inserted into the encrypted file. The insertion position is **KD**.
 - o That random key will be used to generate a PRN.

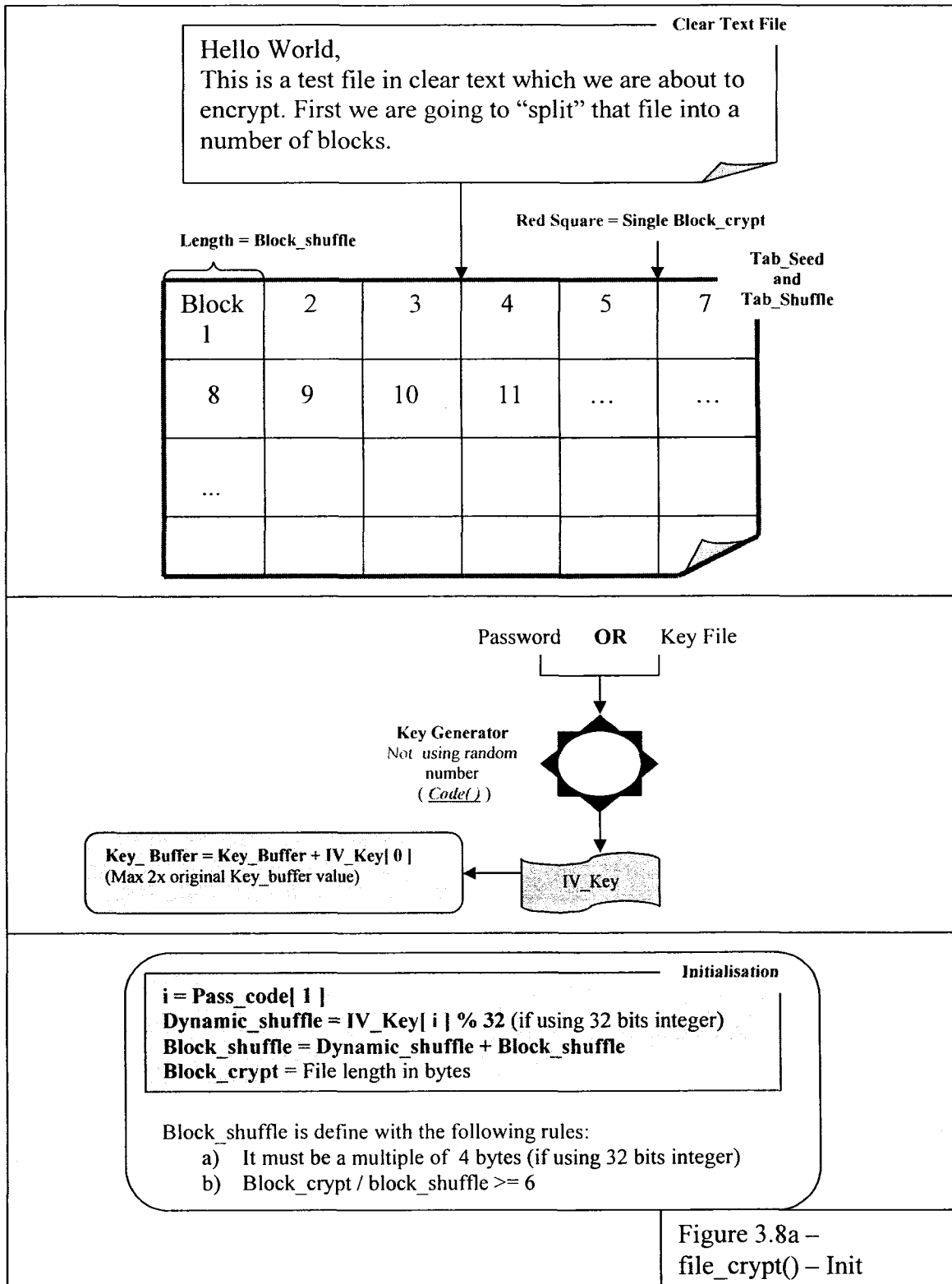
- STEP 2: Seeding
 - A number of keys (**KD**) will be generated and stored into a Key Buffer using a derivation of the initial key generated and the PRN previously generated as the random seed.
 - From that key buffer 2 keys will be selected (**KD**) and an ***AND*** will be conducted. The result is an Encryption Key.
 - A block will be selected from the file virtual array (**KD**) and an ***XOR*** will be conducted with the Encryption Key.
 - A New key will be derived from one of the 2 keys used in creating the Encryption key and replace one of the 2 keys.
 - The process repeats again at STEP 2 until all the blocks have been encrypted (seeded).

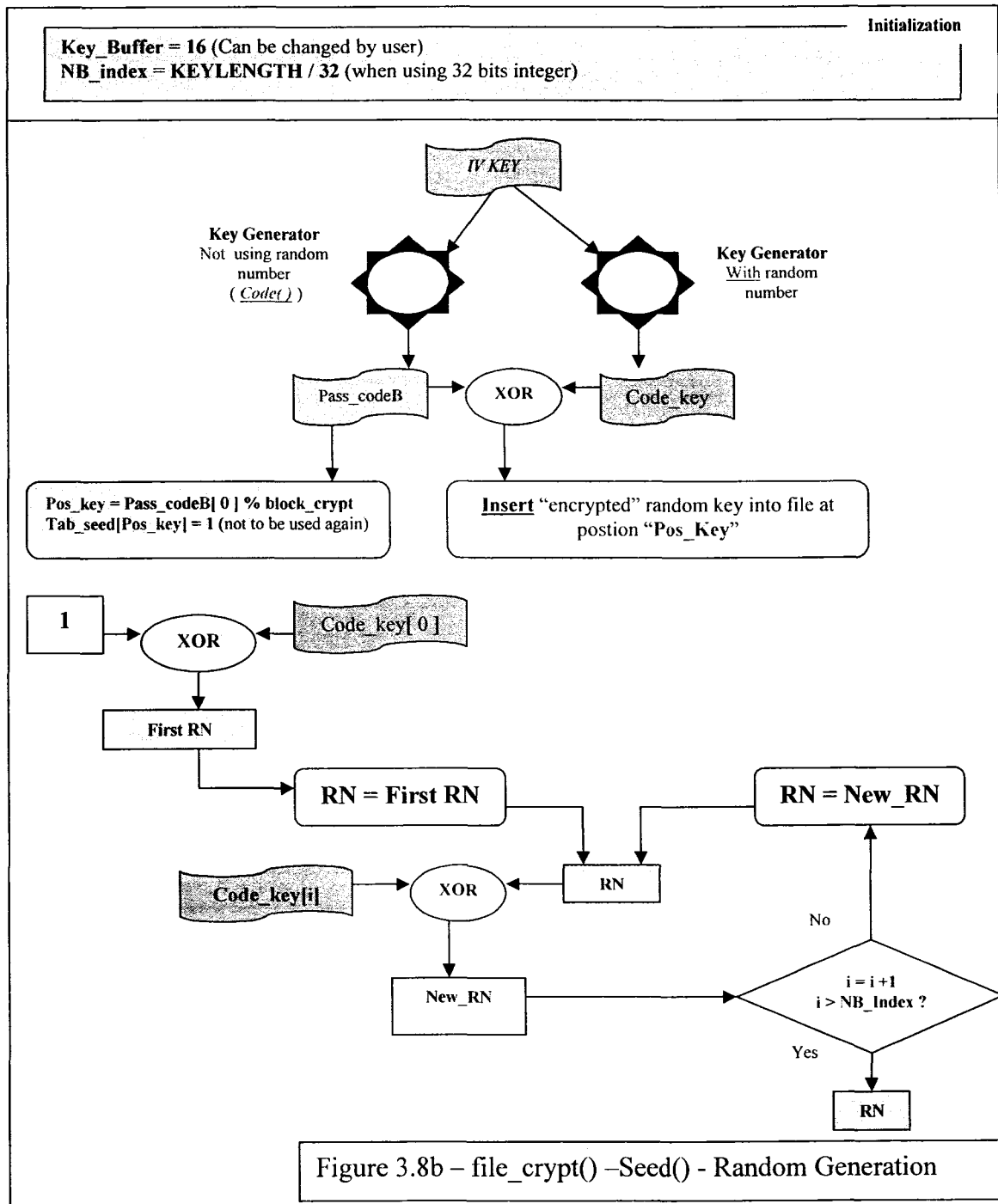
- STEP 3: Shuffling
 - The same virtual array used in STEP 1 will be used again.
 - Two blocks will be selected (**KD**).
 - One out of three possible ***LO*** will be conducted (**KD**) on those two blocks. The result is an Encryption block.
 - A third block will be selected (**KD**) as the block to be encrypted and an ***XOR*** will be conducted with the Encryption Block.
 - One of the two blocks used to generate the Encryption block will then be selected to be the next block to be encrypted (**KD**).

- The process start again at STEP 3 until all the blocks, but the last two, have been encrypted (shuffled).
- The last 2 blocks will be encrypted using an ***XOR*** with 2 new keys generated. Not shuffling the last 2 blocks is required for the decryption process.

3.5 File Encryption Details

Figure 3.8a – file_crypt() – Init and Figure 3.8b – file_crypt() –Seed() - Random Generation





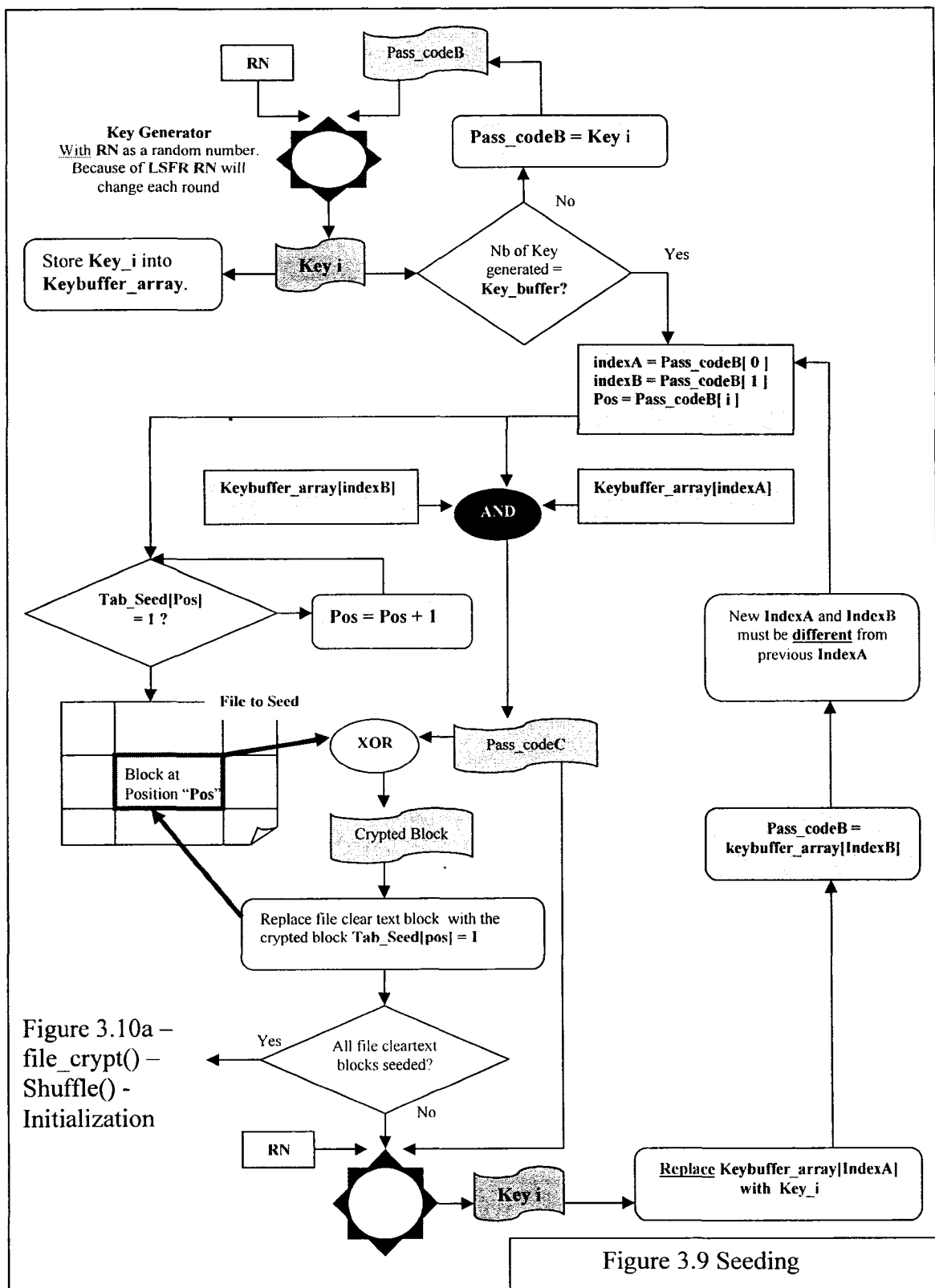
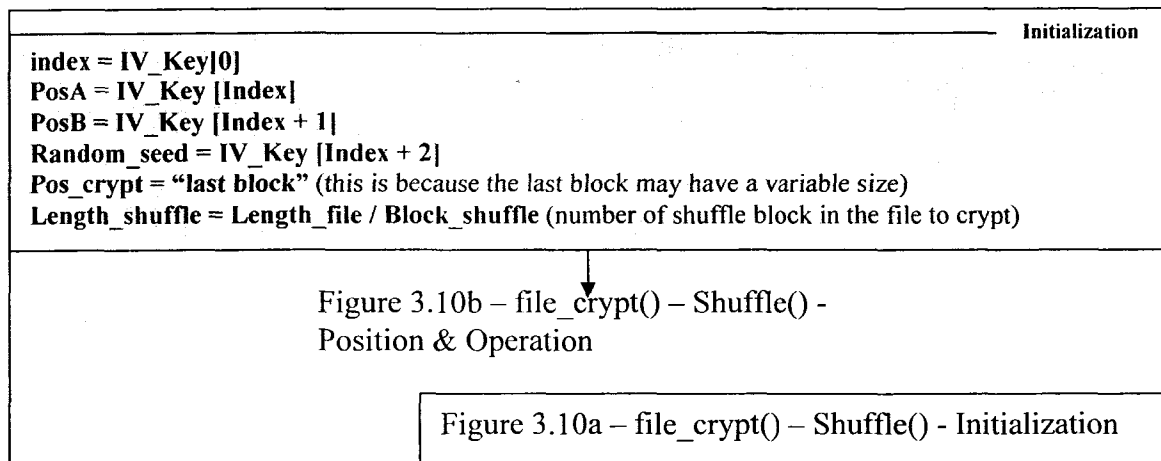


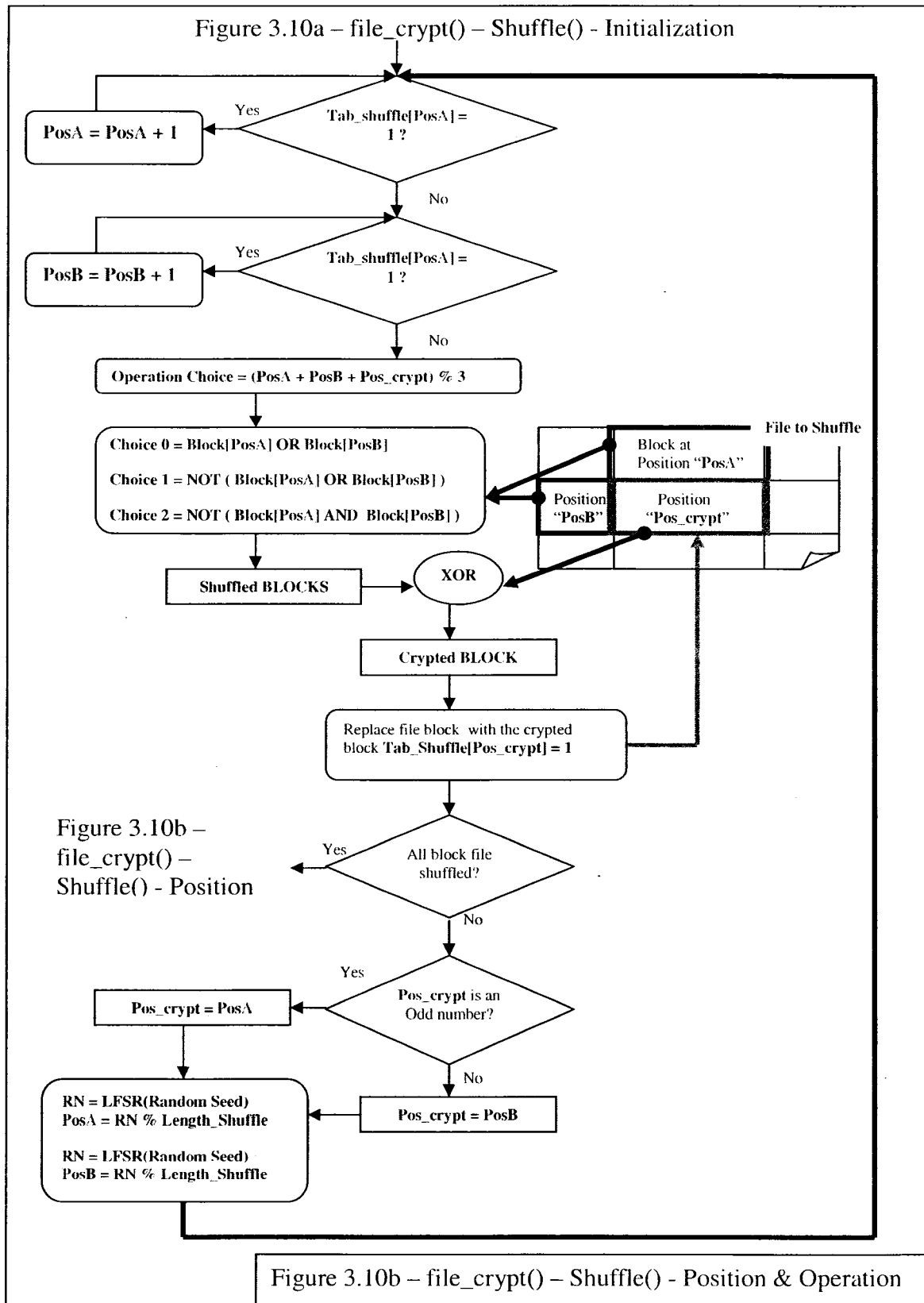
Figure 3.10 Shuffling

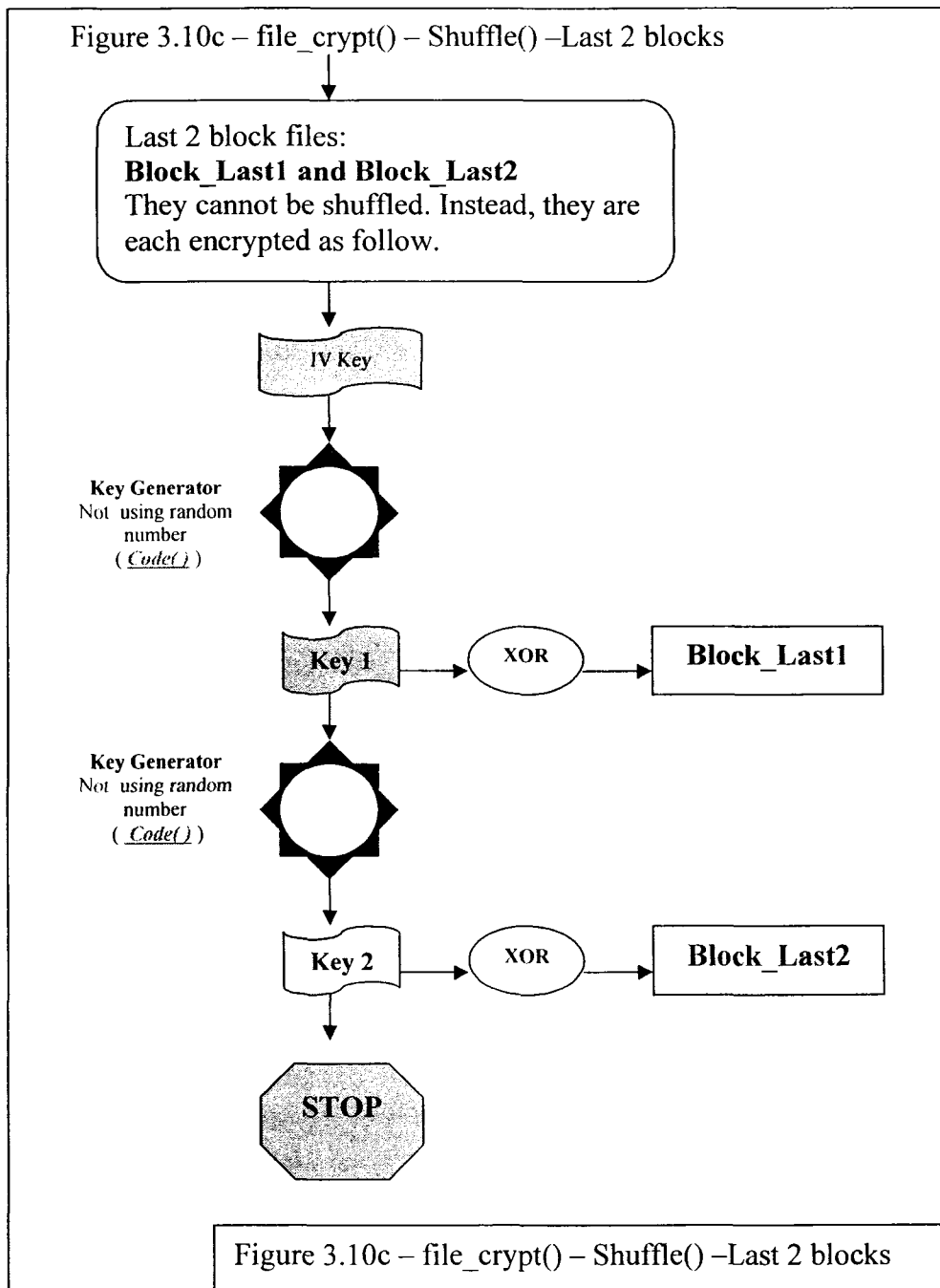
See Figure 3.10a – file_crypt() – Shuffle() - Initialization

And Figure 3.10b – file_crypt() – Shuffle() - Position & Operation

And Figure 3.10c – file_crypt() – Shuffle() –Last 2 blocks

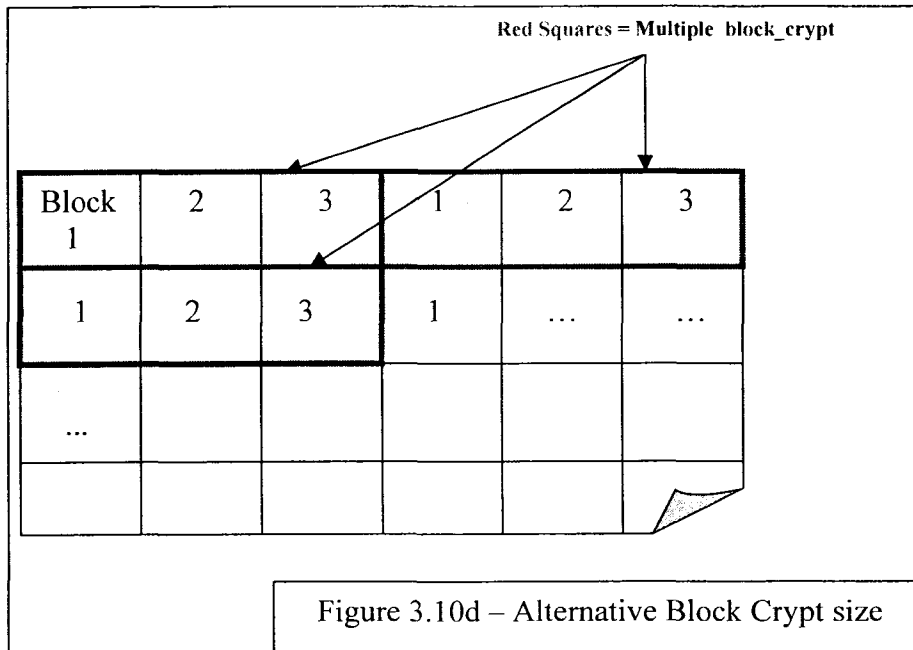






All the KD highlighted in the above steps can be changed to static values, enable or disable. The same is true for the size of the “block_crypt” as shown Figure 3.10d

Alternative Block Crypt size. This means that all the above steps can either be conducted across the entire file or within smaller “working blocks”.



The encryption algorithm provides two methods to encrypt files: Encrypting a file from a password typed by the user that will be transformed into a key, or from a key that has already been generated with the Key generator function, described earlier. If the user wants to use a very long key length, such as 2048 bits, as it is not really possible to remember a long password, he will have to generate a key and store it in a secure area; this key will become the password. The user can also choose to encrypt a file without a password or a key generated with the key generator function. He can simply choose an existing file, and the key will be some of the data in this file (most of the time in the middle of the file). This could be really useful if the user does not want to store a key file.

There are 5 different power levels to encrypt a file.

Seed function

First, a "file array" is defined, if the user uses a 128 bit key length, the algorithm will calculate how many blocks of 128 bits there are in the file that is going to be encrypted. The "file array" will have as many indexes as there are blocks of 128 bits in the file. Each block will match one of the indexes of the "file array". There are only two values that can be found in this array, '0' when the block has not already been encrypted, and '1' when the block has been encrypted.

There is a new feature in the new algorithm:

Many keys will be generated and stored in a buffer. By default, 16 keys are generated. Then the 2 keys will be pseudo randomly selected, mixed together. The key will be used as a filter by doing an Exclusive OR between the data contained in the file and the key. The position of the block file's data (where the filter will be added) is a function of the "pass_clear" string. This is why when we encrypt a file the blocks sequence and the blocks length that is going to be encrypted depends on the password used to encrypt the file. Each time a filter is added, the "file array" is updated to be sure that a filter will not be added twice at the same position.

When a filter is used, a new key is generated. As a result, the key used as a filter is always different and always a function of the previous key that has been generated.

To decrypt a file, since an Exclusive OR has been used to encrypt the file, we simply encrypt the cipher file with the same password again to remove the filters.

Something important is that when the keys are generated it is not possible to add a random number, because it is not possible to find it again, once the key has been added to some data.

This is why with this algorithm, with one password, or key, the user has only one cipher text.

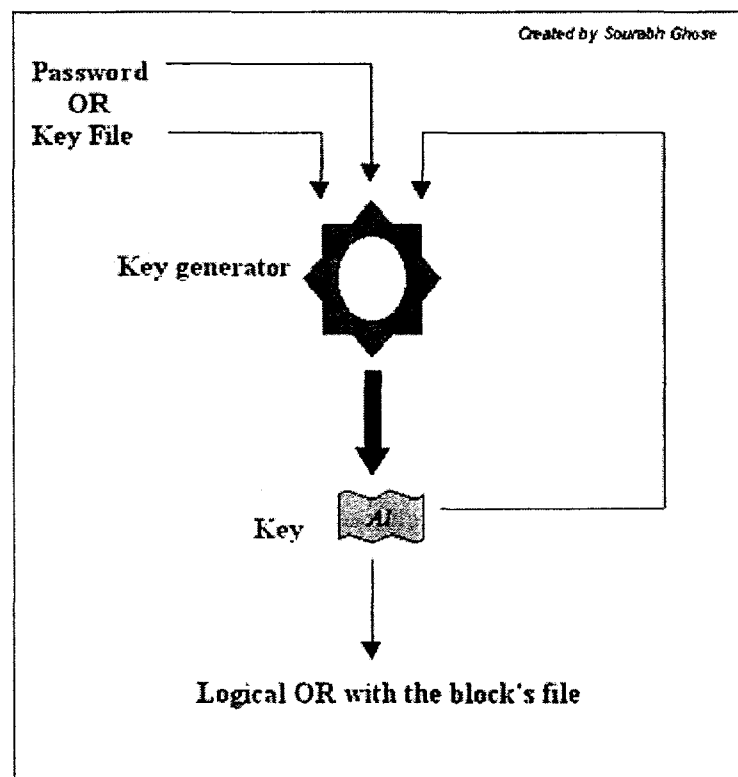


Figure 3.11 File encryption and key generation

Random Seed function

The idea was to be able to have more than one cipher text when the same clear text was encrypted with the same password. This can be achieved by using a random number in the key generation process. This algorithm is nearly the same as the standard one, but this time ONE random number is generated and encrypted using the same

password used to encrypt the clear text. It is then inserted in the cipher text, still at a position function of the "pass_clear" string. After, this random number is used in the key generation. Therefore, with this algorithm for ONE password and ONE clear text there are MANY cipher texts.

To decrypt the cipher text, the random number is extracted from the cipher text first and the cipher text is encrypted again with the password typed by the user and this random number. If it is the right password, the right random number will be extracted, the right keys will be then generated and the algorithm will generate the right clear text.

This algorithm is very strong, indeed if someone tries to decrypt a cipher file, he will never be sure that he has found the right clear file.

Shuffle function

As for the file_seed function we divide the file into blocks (by default block's length = Default integer width = 4 bytes on Linux)

From the last cipher password generated we are going to extract pseudo-random numbers to find 3 file locations:

- position to crypt (pos_crypt) (we always start with the last block)
- position A used in the crypting process (posa)
- position B used in the crypting process (posb)

The algorithm takes the blocks at the position A and B and does a logical operation between them (as a function of the cipher password): OR, NOR, NAND.

Then it adds the result (MASK) with an Exclusive OR to the block at the position pos_crypt.

Then, one of the blocks used to encrypt (A or B) will become the new position to encrypt. The choice between A and B is done by checking if the position that has just been encrypted (pos_crypt) is an odd number or not. If it is then the new encrypt position will be pos A otherwise it will be pos B.

This is done to prevent 2 blocks to be used together more than once to encrypt another block.

Now there are 2 blocks in plain text (in fact they've been "seeded" before) so 2 new cipher passwords are generated, from the one sent as a parameter to this function and is added to the block with an Exclusive OR.

This function is initiated by encrypting the last block of the file. This is because of the way we divide the file into blocks: If the key's length is 128 bits and the file's length is 260 bits then we have got 3 blocks of 128 bits, but the last block only has 4 bits from the file. If we were using this block in the "shuffle" process this would be weak.

Therefore the algorithm does not use it and encrypts it at the start of this function.

The way the algorithm decides which block is going to be encrypted and which blocks are going to be used to generate the MASK is function of the password itself. It uses the value of the password (A) in a LINEAR FEEDBACK SHIFT REGISTER FUNCTION (LFSR). This enables the algorithm to generate many different numbers (B) from (A). An LFSR is really useful to generate a sequence of pseudo random numbers that are always the same if generated from the same initial number.

This function has been taken from the book from Bruce Schneier Applied cryptography Second Edition. The algorithm initializes the LFSR with the password itself. It also uses a different primitive polynomial modulo 2 in function of the length of the "shift register"

the user is using (16, 32 or 64 bits).

This part is really important as the file is then encrypted with its own content.

If we use the seed and shuffle function to encrypt a file, to decrypt we first need to "unshuffle" and then "unseed" the encrypted file.

“UNShuffle” function

The algorithm repeats the process as in the file shuffle function except that it does not make any changes on the file at first. We store the different crypt positions and blocks that are used to create the masks.

We then have 3 arrays:

position[] = position of the block which will be encrypted

crypta[], cryptb[] = position of the 2 blocks used to create the mask

For example:

position[5] = 500

crypta[5] = 232

cryptb[5] = 1300

This means that the 6th block to be encrypted was the block number 500 using the blocks 232 and 1300 to generate the mask.

When the algorithm has filled up these 3 arrays, it then decrypts the last 2 blocks and starts from the end of the array to the beginning to un-shuffle the file.

We can compare that to a "cards castle" once we have done the castle if we want to remove the cards without breaking the castle we need to take out the last card (on the top) and then the next one, etc in a reverse order.

Dynamic Variables

This is one of the prominent features of the algorithm.

The following variables:

- ROUND
- BLOCK SHUFFLE
- BUFFER KEY (used in the seed function, tells the algorithm how many keys to be stored in the buffer).
- MODULO SWAP (which is used in the swap functions and tells the algorithm how big or small can the bits swap process should be.) These variables can be set to be dynamic or static (default = yes).

If set to dynamic, the algorithm will change its value as a function of the password entered. The minimum value is the default value entered for these parameters, and the maximum is double the default value.

For example:

if ROUND = 2, then with the dynamic option ROUND could be a value between 2 and 4.

If ROUND was equal to 10, then it could be a value between 10 and 20.

The MODULO SWAP changes at each round.

All the other Dynamic variables change as block is encrypted.

By default we encrypt a file as one big block, so these values will change only once.

CHAPTER IV

ALGORITHM APPLICATIONS

Two applications using the cryptography algorithm were first created for UNIX (such as Linux, SunOS, Silicon Graphics, HPUX, etc.). The reasons for this choice were because this system is the most used in the computer world. Also, because no graphical user interface needed to be created, these applications could be created faster.

These applications have been created using the C language. They all use a library which contains all the cryptography functions created for this project and only consist of a user interface for the use of this library. Particular attention has been given to the error check in these applications. Even if they are badly used, an understandable error message is displayed. The following applications have been designed:

Encrypt/Decrypt file application:

This application is used to encrypt or decrypt a file. Some parameters have to be set by the user: the password, the file to encrypt/decrypt, the destination file, the length of the key, the encrypting power method, the round of the key generator used. The user can even specify a custom crypt's block length and a custom shuffle's block length. There is also an interactive mode where the application is prompting you for each parameter required. This is a security enhancement if the user does not want someone doing a 'ps' looking at his parameters

Secure chat, similar to the 'talk' UNIX command:

Secure chat, similar to the 'talk' UNIX command:

This application can be used to have a secure conversation over a network (Internet for instance). It uses a stream encryption method.

Key generator application:

This application is used to generate a key which will be stored in a file. Some parameters have to be set by the user: the password or the random generation flag, the destination file and the key length to be generated.

Login application similar to the UNIX system:

This application, is used when a user wants to log onto the system. Parameters must be set by the user, such as his password and the location of the password database which contains his cipher password.

Password management application:

This application is quite similar to the key generator application but the key generated is stored into a password database with the user login name. Some parameters have to be set by the user such as the user name and his new password. If the user already has a cipher password stored into the password database, he will have to re-type his old password to be able to change it with the new one.

Hide/Extract engine application:

This application is used to hide a file in another file or to extract a previously hidden file. The user can choose to hide a file at the beginning or at the end of another file. This is a completely separate part of the cryptography called steganography. It will be too long to create a strong steganography algorithm so this is why it is only a simple algorithm (Added to the beginning or at the end of a file). This part will be done, because

even if the algorithm is simple, it could be useful. This is because, with this algorithm, if the user wants to hide a file into a picture, video or sound file, it becomes invisible.

CHAPTER V

ALGORITHM ANALYSIS

This section evaluates how the DCA successfully evades two powerful symmetric-key block cipher cryptanalysis techniques: Linear Cryptanalysis [3] and Differential Cryptanalysis [4].

5.1 Linear Cryptanalysis

Introduction

Linear Cryptanalysis tries to take advantage of high probability occurrences of linear expressions involving plaintext bits, "ciphertext" bits (actually we use bits from the 2nd last round output), and subkey bits. It is a known plaintext attack: that is, it is premised on the attacker having information on a set of plaintexts and the corresponding ciphertexts. However, the attacker has no way to select which plaintexts (and corresponding ciphertexts) are available. In many applications and scenarios it is reasonable to assume that the attacker has knowledge of a random set of plaintexts and the corresponding ciphertexts.

The idea is that, for each candidate subkey, we partially decrypt the cipher and check if the relation holds. If the relation holds then increment its corresponding counter. At the end, the candidate key that counts furthest from $\frac{1}{2}$ is the most likely subkey.

The Piling up Lemma:

Suppose X_1, X_2, \dots are independent random variables from $\{0,1\}$. And

$$\Pr[X_i = 0] = p_i, \quad i = 1, 2, \dots \text{Hence,}$$

$$\Pr[X_i = 1] = 1 - p_i, \quad i = 1, 2, \dots$$

The independence of X_i, X_j implies

$$\Pr[X_i = 0, X_j = 0] = p_i p_j$$

$$\Pr[X_i = 0, X_j = 1] = p_i (1 - p_j)$$

$$\Pr[X_i = 1, X_j = 0] = (1 - p_i) p_j$$

$$\Pr[X_i = 1, X_j = 1] = (1 - p_i)(1 - p_j)$$

Let $\mathcal{E}_{i_1, i_2, \dots, i_k}$ denote the bias of $X_{i_1} \oplus \dots \oplus X_{i_k}$

$$\text{Then } \mathcal{E}_{i_1, i_2, \dots, i_k} = 2^{k-1} \prod_{j=1}^k \mathcal{E}_{i_j}.$$

Analysis of DCA

Consider a function f that takes an 8-bit input (x) and an 8-bit subkey (k) as input and produces an 8 bit output (y). We can write this as $y = f(x, k) \pmod{2}$. Imagine that DCA had been designed in such a way that we could write the function f as a linear combination of x and k modulo 2. That is, what if the function f were designed as $y = f(x, k) = Mx + Dk \pmod{2}$ where M and D are constant 8×8 matrices. The function f would look like this:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} & M_{0,4} & M_{0,5} & M_{0,6} & M_{0,7} \\ M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} & M_{1,4} & M_{1,5} & M_{1,6} & M_{1,7} \\ M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} & M_{2,4} & M_{2,5} & M_{2,6} & M_{2,7} \\ M_{3,0} & M_{3,1} & M_{3,2} & M_{3,3} & M_{3,4} & M_{3,5} & M_{3,6} & M_{3,7} \\ M_{4,0} & M_{4,1} & M_{4,2} & M_{4,3} & M_{4,4} & M_{4,5} & M_{4,6} & M_{4,7} \\ M_{5,0} & M_{5,1} & M_{5,2} & M_{5,3} & M_{5,4} & M_{5,5} & M_{5,6} & M_{5,7} \\ M_{6,0} & M_{6,1} & M_{6,2} & M_{6,3} & M_{6,4} & M_{6,5} & M_{6,6} & M_{6,7} \\ M_{7,0} & M_{7,1} & M_{7,2} & M_{7,3} & M_{7,4} & M_{7,5} & M_{7,6} & M_{7,7} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \\
+ \begin{bmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} & D_{0,4} & D_{0,5} & D_{0,6} & D_{0,7} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} & D_{1,4} & D_{1,5} & D_{1,6} & D_{1,7} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} & D_{2,4} & D_{2,5} & D_{2,6} & D_{2,7} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} & D_{3,4} & D_{3,5} & D_{3,6} & D_{3,7} \\ D_{4,0} & D_{4,1} & D_{4,2} & D_{4,3} & D_{4,4} & D_{4,5} & D_{4,6} & D_{4,7} \\ D_{5,0} & D_{5,1} & D_{5,2} & D_{5,3} & D_{5,4} & D_{5,5} & D_{5,6} & D_{5,7} \\ D_{6,0} & D_{6,1} & D_{6,2} & D_{6,3} & D_{6,4} & D_{6,5} & D_{6,6} & D_{6,7} \\ D_{7,0} & D_{7,1} & D_{7,2} & D_{7,3} & D_{7,4} & D_{7,5} & D_{7,6} & D_{7,7} \end{bmatrix} \begin{bmatrix} k_0 \\ k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \\ k_7 \end{bmatrix} \pmod{2}$$

To do this we would only have to change the seeding and shuffling functions to linear functions. All XOR's are already linear functions. For example, and XOR can be written like $z = i(x, y)$:

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \pmod{2}$$

So if the seeding and shuffling functions were linear equations we could easily find a linear function $y = f(x, k) \pmod{2}$.

Assume the seeding and shuffling functions are linear function, which we can write as $y = g(x) = Ex$ where E is a constant 8×8 matrix:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \pmod{2}$$

DCA has two rounds of the function f (one for each of two keys generated from the key buffer $K1$ and $K2$). So if P is the plaintext and C is the ciphertext, then:

$$\begin{aligned} C &= f(g(f(P, K1)), K2) \\ &= M \times g(f(P, K1)) + D \times K2 \\ &= E \times M \times f(P, K1) + D \times K2 \\ &= E \times M \times (M \times P + D \times K1) + D \times K2 \\ &= E \times M^2 \times P + E \times M \times D \times K1 + D \times K2 \pmod{2} \end{aligned}$$

Now define three new constant 8×8 matrices: $R = E \times M^2$, $S = E \times M \times D$, and $T = D$.

Even if we use independent subkeys for $K1$ and $K2$, we cannot have a linear equation.

If in worst case, bit 0 of the ciphertext is equal to bit 3 of the plaintext XORed with bit 4 of the plaintext XORed with bit 5 of the plaintext XORed with bit 0 of $K1$, etc. We would have similar equations for bits 1 through 7 of the ciphertext. So for a known ciphertext attack with one plaintext-ciphertext pair we would have 8 equations and 32 unknowns, which does not do much good.

But with 4 plaintext-ciphertext pairs you would have 32 equations and 32 unknowns. Using Gaussian elimination or Cramer's rule it is easy to see that we could solve this system with something on the order of 32 calculations which is no good again.

Steps for performing linear cryptanalysis on DCA

1] We need several good linear approximations for each linear function. They should hold for more than 50% of the possible inputs. Each equation is just XOR's modulo 2 (equivalent to XORing them all together) to get an answer of either 0 or 1.

2] We need about 100 plaintext/ciphertext pairs. They don't have to be chosen plaintext, this is a known plaintext attack.

3] Do step 4 for every possible subkey

4] For each plaintext/ciphertext pair that we have

4.1] Find Q, the output of the seeding.

4.2] Find S, the output of shuffling.

4.3] See if each of the linear approximations hold for S as input and Q as output. We don't really care what the first key is because any bit in the second key has a 50% chance of being a 0 and not touching anything. If it were a 1 then it would change one of our S bits but we wouldn't really care because our linear approximations are still biased away from 50%. If S was chosen well then we would expect S and Q to show bias with the linear approximations. If S is not anything like the input that was really used when the ciphertext/plaintext pair was generated then we're just plugging in random bits for S and Q and we would expect our linear approximations to hold about 50% of the time. The subkey that we guess that shows the highest deviation away from 50% is the one most likely to be the real key. The level of effort to break the cipher then comes from the size of the subkey and not the key size.

Linear cryptanalysis won't produce such dramatic results on DCA, because DCA does not have many linear functions like S-P networks. The level of effort to do linear

cryptanalysis on DCA is still dependent on the size of the subkey, but we need a lot of plaintext/ciphertext pairs which makes it pretty much infeasible.

5.2 Differential Cryptanalysis

Introduction

Differential cryptanalysis exploits the high probability of certain occurrences of plaintext differences and differences into the last round of the cipher. For example, consider a system with input $X = [X_1 X_2 \dots X_n]$ and output $Y = [Y_1 Y_2 \dots Y_n]$. Let two inputs to the system be X' and X'' with the corresponding outputs Y' and Y'' , respectively. The input difference is given by $\Delta X = X' \oplus X''$ where " \oplus " represents a bit-wise exclusive-OR of the n -bit vectors and, hence,

$$\Delta X = [\Delta X_1 \Delta X_2 \dots \Delta X_n]$$

The main difference from linear attack is that differential attack involves comparing the XOR of two inputs to the XOR of the corresponding outputs. Differential attack is a chosen-plaintext attack. We consider inputs x and x^* having a specified XOR value. We decrypt y and y^* using all possible key and determine if their XOR has a certain value. Whenever it does, increment the corresponding counter. At the end, we expect the largest one is the most likely subkey.

In order to use differential cryptanalysis on DCA the attacker would require large amounts of plaintexts. The number of chosen plaintexts needed is dependent on the ability of XOR differences to propagate through a block cipher algorithm (and therefore partly dependent on the number rounds) and the computational effort comes from the size

of the subkey used for each round. The search space of differential cryptanalysis is the subkey of the last round and not the encryption key.

Steps for performing Differential Cryptanalysis on DCA

1] Choose an XOR difference for the plaintext pairs

Let us go ahead and use an XOR difference of “10000000.” This means that each of our plaintext pairs will differ in only the first bit. For example “00000000” will be paired with “10000000”, “00000001” will be paired with “10000001”, ..., “01100100” will be paired with “11100100”, ..., and “01111111” will be paired with “11111111”.

2] Generate ciphertext pairs for some plaintext pairs

Differential cryptanalysis is a chosen plaintext attack, so we will need access to the encryption equipment with the key installed. This doesn't mean we can see the key, though we just need to have access to the equipment.

Now we are going to encrypt 25 randomly chosen plaintext pairs from our plaintext pairs above, for a total of 50 encryptions. If our cipher had more rounds we would have to do a lot more work.

When we do these 50 encryptions we get 25 ciphertext pairs, paired by the fact that both ciphertexts in a pair came from one of the plaintexts of a plaintext pair.

Plaintext -> Ciphertext

One pair:

01111001 -> 11111001

11111001 -> 11111110

Another pair:

01000100 -> 10101100

11000100 -> 10011001

A third pair:

00000011 -> 01000010

10000011 -> 10110111

etc.

3] Get 2 pieces of information out of each ciphertext pair

3.1] The first piece of information we want out of each ciphertext pair is the input to seeding function.

3.2] The second piece of information we need is not so trivial. We are going to have to find out what the expected XOR difference that came out of the shuffling function.

4] Try all possible values for the subkey, which is extremely hard with larger keys.

For every subkey, what we have to do is recreate seeding, the 8-bit XOR after seeding, and the shuffling for each ciphertext pair. The reason we have to use 25 ciphertext pairs instead of just one is that there is a chance a false input might get lucky and produce the right output, but this won't happen 25 times. The number 25 has nothing to do with the key size or block size so it does not add to our measure of complexity.

Each step has extremely large complexity due to which differential cryptanalysis does not produce dramatic results against the DCA.

5.3 Performance Analysis

Here are speed benchmarks for some of the most common cryptography algorithms compared to the DCA. The test was run on a Pentium 4 2.1 GHz processor on Windows XP professional platform. As shown, the DCA is considerably faster than its peers.

Algorithm	Megabytes(2^{20} bytes) Processed	Time Taken	MB/Seconds
DCA	256	3.841	66.649
MD5	1.02e+003	4.726	216.674
Ripemd-160	256	4.867	52.599
SHA-512	64	5.618	11.392
HMAC	1.02e+003	4.726	216.674
AES(256)	256	5.308	48.229
DES	128	5.998	21.340
RC4	512	4.517	113.350

Table 5.1 Speed Comparison of Cryptography Algorithms

Analysis of DCA, AES, 3DES and RC2

For the tests, the Microsoft Application Center Test (ACT) was used, which is designed to stress test. Application Center Test can simulate a large group of users by opening multiple connections to the server and rapidly sending HTTP requests. It also allows us to build realistic test scenarios where we can call the same method with a

allows us to build realistic test scenarios where we can call the same method with a randomized set of parameter values. This is an important feature, whereby users are not expected to call the same method with the same parameter values over and over again. The other useful feature is that Application Center Test records test results that provide the most important information about the performance of the Web application.

The algorithms were used to encrypt and decrypt data. Tests were performed with a data size of 4 KB, 100 KB, and 500 KB to see how the size of data impacts performance.

Figure 3.12a Plot of Request per second against user load, data = 4 KB

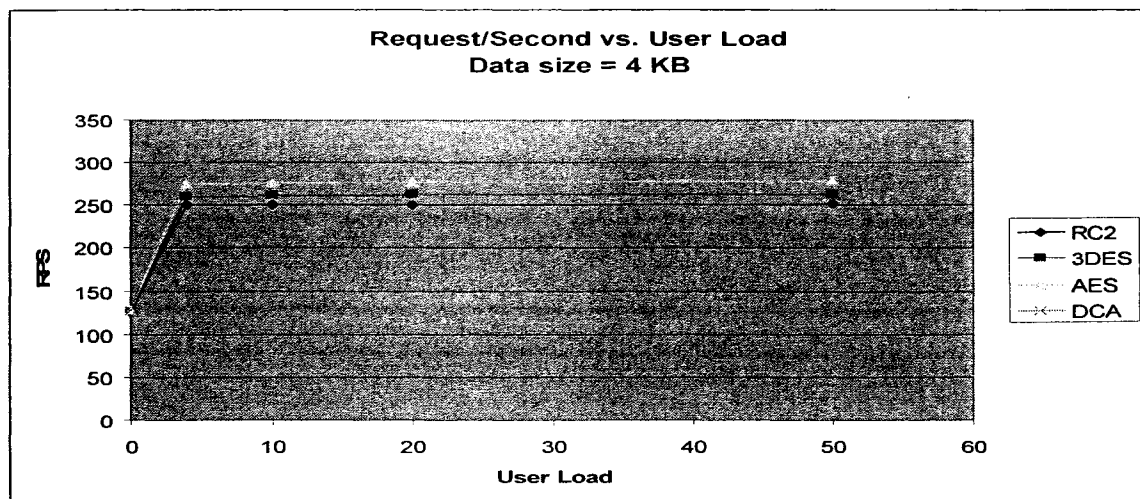
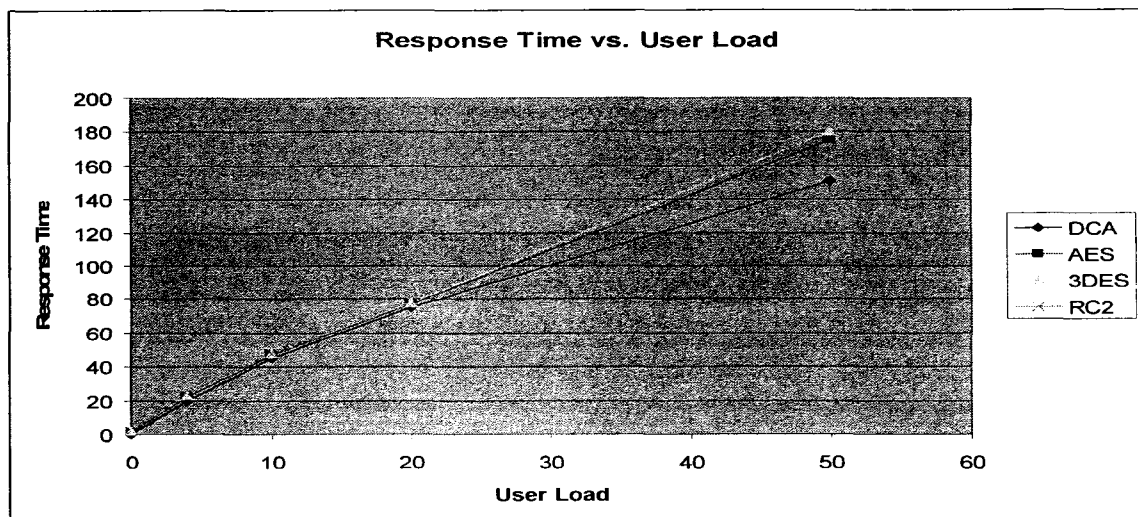


Figure 3.12b Plot of Response time against user load, data = 4 KB



With small data, we find that the DCA is the fastest of all the methods. A key length of 256 bits was chosen.

Next was the AES (Advanced Encryption standard). It has a variable block length and key length, which may be chosen to be any of 128, 192, or 256 bits. It also has a variable number of rounds to produce the cipher text, which depends on the key length and the block length.

Next was the Triple DES (3DES) was invented to improve the security of DES by applying DES encryption three times using three different keys (note that encrypting data three times with the same key does not offer any value). It is simply another mode of DES, but it is highly secure and therefore slower in performance. It takes a 192-bit key, which is broken into three 64-bit subkeys to be used in the encryption procedure. The procedure is exactly like DES, but it is repeated three times, making it much more secure. The data is encrypted with the first subkey, decrypted with the second subkey, and encrypted again with the third subkey.

RC2 turns out to be the slowest method when the data being encrypted is small. It has an expensive computation up front to build a key-dependent table, which apparently is high compared to the cost of encrypting small data.

Figure 3.13a Plot of Request per second against user load, data = 100 KB

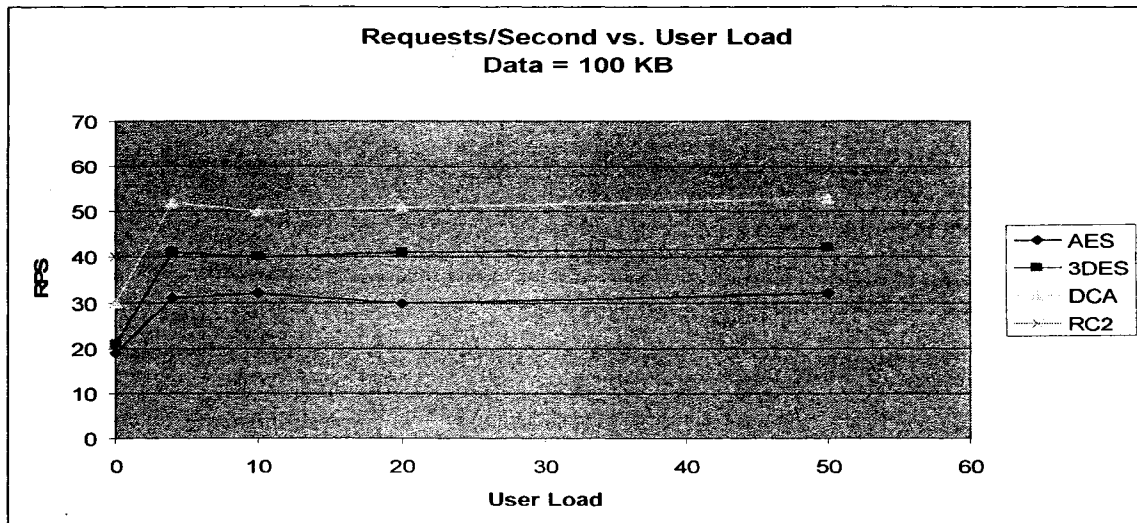
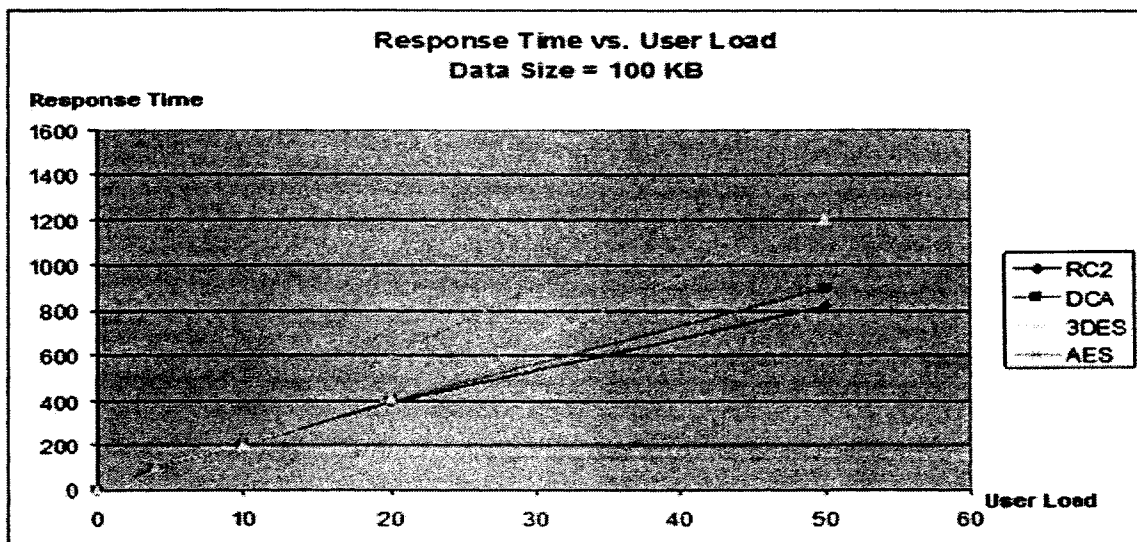


Figure 3.13b Plot of Response time against user load, data = 100 KB



By increasing the size of data being encrypted and decrypted, we see an entirely different picture to what we saw in the previous test. RC2 is the fastest, followed by DCA, which is around 20% faster than 3DES. Note that the expensive computation in RC2 to build the key-dependent table is amortized over more data. AES in this case is the slowest; 25% slower than 3DES. Note that we are using a 256-bit key for AES encryption, which makes it stronger than the other methods (though there has been some press about possible attacks against AES, which might be better than brute force attack) and for the same reason the slowest of all. Similarly, we used a 192-bit key in case of 3DES. Using a same-length key does not necessarily mean that different algorithms will have the same strength. Different algorithms have different characteristics and hence they may not provide the same strength.

There is always a tradeoff between security and performance. We need to understand the value of sensitive data, the deployment cost, and usability/performance tradeoffs before you can begin choosing a right algorithm for securing data. If the cost of data that is being protected is high, then we must consider taking a performance hit to secure the data. Otherwise, we may be better off using a less secure algorithm.

Figure 3.14a Plot of Request per second against user load, data = 500 KB

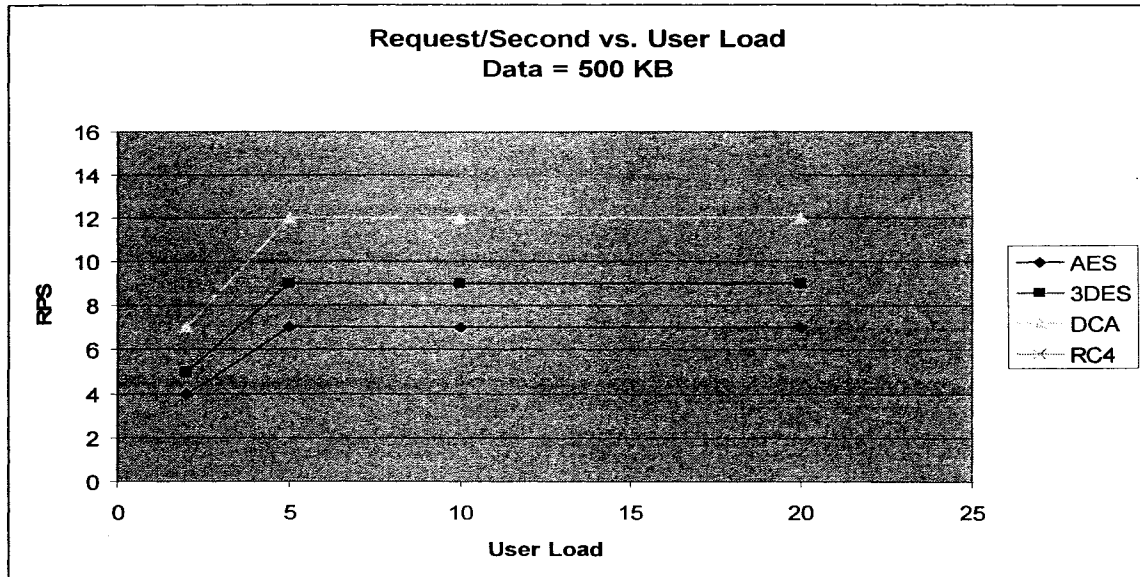
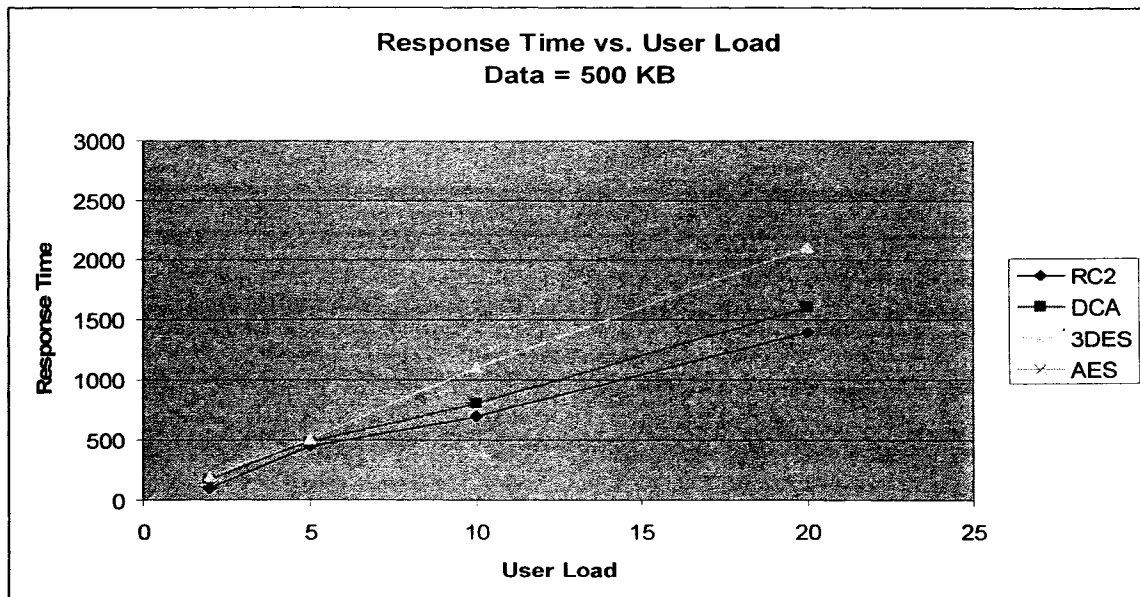


Figure 3.14b Plot of Response time against user load, data = 500 KB



With the increasing size of data being encrypted and decrypted, we see the same trend prevailed in this test too.

CHAPTER VI

CONCLUSION

The main problem in this algorithm concept was that it was necessary to always have in mind the following words: security, efficiency and portability. Everything has been done to respect these keywords.

To conclude, here is a summary of the algorithm specifications originally. All initial design goals were met successfully.

- Private Key algorithm
- The source code can be public without making the algorithm weak
- Dynamic Algorithm
- Pseudo Random numbers generated using a Linear Feedback Shift Register and ISAAC Algorithm
- Direct disk access or Memory buffer method in order to run on low specification computer and Palm devices
- Stream encryption option
- Block encryption option
- Key generator: Bits manipulations, bilateral bits swapping and logical bits operations, random generator.
- Encrypt file (using the key generator): Variable length cipher block algorithm.
- 5 Power levels: Seed, Random Seed, Shuffle, Seed + Shuffle, Random Seed + Shuffle.

- Key length unlimited (as big as your integer type can take).

The algorithm was successfully able to evade two of the most powerful cryptanalysis techniques: Linear and Differential Cryptanalysis.

As proposed, the following applications we designed too, highlighting some of the applications of the cryptography suite:

- Encrypt/Decrypt file application
- Secure chat
- Key generator application
- Login application
- Password management application
- Hide/Extract engine.

The project applications were tested on many different operating systems (OS): Windows 9x, Windows NT, UNIX (Linux, HPUX, Silicon Graphics, and Solaris). Because each operating system is different (memory management, permission rights, variable management, etc) these tests were very important. For example, a problem regarding the way the password was stored in the memory occurred once. No errors were generated when the application was running on Linux, but on the Silicon graphics OS an error was generated. This was because on Linux the memory management is less strict than on Silicon Graphics (or at least at the time on the Linux OS there were not as many users as there were on SGI). Therefore, sometimes the application worked fine on the Silicon graphics and Windows but not on the SunOS because these first two OS were more compliant in certain areas. These tests helped in eliminating several bugs.

The algorithm has reached its maturity after several iterations and careful cryptanalysis. Future work includes designing attack cases against the algorithm.

REFERENCES

1. National Bureau of Standards, Data Encryption Standard, U.S. Department of Commerce, FIPS Publication 46, Jan 1977.
2. M.J. Weiner, "Efficient DES Key Search," Advances in Cryptology--CRYPTO '93 Proceedings, Springer-Verlag, in preparation.
3. M. Matsui, "Linear Cryptanalysis Method for DES Cipher," Advances in Cryptology--CRYPTO '93 Proceedings, Springer-Verlag, 1994, in preparation.
4. E. Biham and A. Shamir, Differential Cryptanalysis of the Data Encryption Standard, Springer-Verlag, 1993.
5. R.C. Merkle, "Fast Software Encryption Functions," Advances in Cryptology--CRYPTO '90 Proceedings, Springer-Verlag, 1991, pp. 476-501.
6. R.C. Merkle, "Method and Apparatus for Data Encryption," U.S. Patent 5,003,597, 26 Mar 1991.
7. T.W. Cusick and M.C. Wood, "The REDOC-II Cryptosystem," Advances in Cryptology--CRYPTO '90 Proceedings, Springer-Verlag, 1991, pp. 545-563.
8. M.C. Wood, "Method of Cryptographically Transforming Electronic Digital Data from One Form to Another," U.S. Patent 5,003,596, 26 Mar 1991.
9. B. Schneier, Applied Cryptography, John Wiley & Sons, New York, 1994.
10. X. Lai, J. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," Advances in Cryptology--EUROCRYPT '91 Proceedings, Springer-Verlag, 1991, pp. 17-38.

11. J.L. Massey and X. Lai, "Device for Converting a Digital Block and the Use Thereof," International Patent PCT/CH91/00117, 16 May 1991.
12. J.L. Massey and X. Lai, "Device for the Conversion of a Digital Block and Use of Same," U.S. Patent 5,214,703, 25 May 1993.
13. RSA Laboratories, Answers to Frequently Asked Questions About Today's Cryptography, Revision 2.0, RSA Data Security Inc., 5 Oct 1993.
14. National Institute of Standards and Technology, "Clipper Chip Technology," 30 Apr 1993.
15. Nicolas Courtois and Josef Pieprzyk, Cryptanalysis of Block Ciphers with Overdefined Systems of Equations, 9 Nov 2002
16. Ron Rivest, "The MD5 Message-Digest Algorithm," RFC 1321, April 1992.
17. Bruce Schneier, "Section 18.5 MD5," Applied Cryptography, Second Edition, John Wiley & Sons, 1996.
18. Thomas Berson, "Differential Cryptanalysis Mod 2^{32} with Applications to MD5," Advances in Cryptology - EuroCrypt '92 Proceedings, Volume 658 of Lecture Notes in Computer Science, Springer-Verlag.
19. RSA Laboratories Security Bulletin #4, <ftp://ftp.rsa.com/pub/pdfs/bulletn4.pdf>
20. Hans Dobbertin, Cryptanalysis of MD5 Compress, <http://www-cse.ucsd.edu/~bsy/dobbertin.ps>
21. Hans Dobbertin, Antoon Bosselaers, Bart Preneel, RIPEMD-160: A Strengthened Version of RIPEMD. A joint publication by the German Information Security Agency (POB 20 03 63, D-53133 Bonn, Germany) and the Katholieke Universiteit Leuven,

- ESAT-COSIC (K. Mercierlaan 94, B-3001 Heverlee, Belgium), 18 April 1996.
- Available from <http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html>
22. Hans Dobbertin, Antoon Bosselaers, Bart Preneel, The RIPEMD-160 page,
<http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html>
23. A. Menezes, P.C. van Oorschot, S.A. Vanstone, "Algorithm 9.55 RIPEMD-160 hash function," Handbook of Applied Cryptography, CRC Press, 1997.
<http://www.cacr.math.uwaterloo.ca/hac/about/chap9.pdf>, .ps
24. ISO/IEC 10118-3:1998, Information technology -- Security techniques -- Hash-functions -- Part 3: Dedicated hash-functions.
25. U.S. National Institute of Standards and Technology, FIPS 180-2, Secure Hash Standard (SHS). <http://csrc.nist.gov/encryption/tkhash.html>
26. M. Bellare, R. Canetti, H. Krawczyk, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, February 1997.
27. M. Bellare, R. Canetti, H. Krawczyk, "Keying hash functions for message authentication," Extended abstract in Advances in Cryptology - CRYPTO '96 Proceedings, Volume 1109 of Lecture Notes in Computer Science (N. Koblitz, ed.), Springer-Verlag, 1996.
28. M. Bellare, R. Canetti, H. Krawczyk, "Message authentication using hash functions: The HMAC construction," *RSA Laboratories' CryptoBytes* vol. 2, no. 1, Spring 1996.
29. P. Cheng, R. Glenn, "Test Cases for HMAC-MD5 and HMAC-SHA-1," RFC 2202, September 1997.
30. J. Kapp, "Test Cases for HMAC-RIPEMD160 and HMAC-RIPEMD128," RFC 2286, February 1998.

31. NIST, AES Home Page, <http://www.nist.gov/aes/>
32. AES Round 1 Information, <http://csrc.nist.gov/encryption/aes/round1/round1.htm>
33. AES Round 2 Information, <http://csrc.nist.gov/encryption/aes/round2/round2.htm>
34. The CAESAR - Candidate AES for Analysis and Reviews project,
<http://www.dice.ucl.ac.be/crypto/CAESAR/caesar.html>
35. Lars Knudsen, Vincent Rijmen, *The Block Cipher Lounge - AES*,
<http://www.iu.uib.no/~larsr/aes.html>
36. John Savard, Towards the 128-bit Era - AES Candidates,
<http://fn2.freenet.edmonton.ab.ca/~jsavard/crypto/co0408.htm>
37. Eli Biham, "A Note on Comparing the AES Candidates," Presented at the 2nd AES Conference. <http://csrc.nist.gov/encryption/aes/round1/conf2/papers/biham2.pdf>
38. Olivier Baudron, Henri Gilbert, Louis Granboulan, Helena Handschuh, Antoine Joux, Phong Nguyen, Fabrice Noilhan, David Pointcheval, Thomas Pornin, Guillaume Poupard, Jacques Stern, Serge Vaudenay, "Report on the AES Candidates," Presented at the 2nd AES Conference.
39. G. Carter, E. Dawson, L. Nielsen, "Key Schedule Classification of the AES Candidates," Presented at the 2nd AES Conference.
40. B. Preneel, A. Bosselaers, V. Rijmen, B. Van Rompay, L. Granboulan, J. Stern, S. Murphy, M. Dichtl, P. Serf, E. Biham, O. Dunkelman, V. Furman, F. Koeune, G. Piret, J-J. Quisquater, L. Knudsen, H. Raddum, "Comments by the NESSIE Project on the AES Finalists," Submitted to NIST as an AES comment, May 2000.
41. Thomas S. Messerges, "Securing the AES Finalists Against Power Analysis Attacks," Presented at *Fast Software Encryption* 2000, New York.

42. Nicolas Courtois, Josef Pieprzyk, "Cryptanalysis of Block Ciphers with Overdefined Systems of Equations". pp267–287, ASIACRYPT 2002.
43. U.S. National Institute of Standards and Technology, NIST FIPS PUB 46-2 (supercedes FIPS PUB 46-1), "Data Encryption Standard", U.S. Department of Commerce, December 1993.
44. U.S. National Institute of Standards and Technology, NIST FIPS PUB 74, "Guidelines for Implementing and Using the NBS Data Encryption Standard".
45. Bruce Schneier, "Chapter 12 Data Encryption Standard," Applied Cryptography, Second Edition, John Wiley & Sons, 1996.
46. A. Menezes, P.C. van Oorschot, S.A. Vanstone, "Section 7.4 DES," Handbook of Applied Cryptography, CRC Press, 1997.
47. Eli Biham, Adi Shamir, "Differential Cryptanalysis of the Full 16-Round DES," CS 708, Proceedings of CRYPTO '92, Volume 740 of Lecture Notes in Computer Science, December 1991.
48. Eli Biham, Adi Shamir, "Differential cryptanalysis of DES-like cryptosystems," Technical report CS90-16, Weizmann Institute of Science. Advances in Cryptology - CRYPTO '90 Proceedings and Journal of Cryptology, Vol. 4, No. 1, pp. 3-72, 1991.
49. Eli Biham, Adi Shamir, Differential Cryptanalysis of the Data Encryption Standard, Springer-Verlag, 1993.
50. M. Matsui, "Linear cryptanalysis method for DES cipher," Advances in Cryptology - EUROCRYPT '93 Proceedings, Volume 765 of Lecture Notes in Computer Science (T. Hellese, ed.), pp. 386-397. Springer-Verlag, 1994.

51. M. Matsui, "The First Experimental Cryptanalysis of the Data Encryption Standard,"
Advances in Cryptology - CRYPTO '94 Proceedings, Volume 839 of Lecture Notes
in Computer Science, Springer-Verlag, 1994.
52. M. Matsui, "On Correlation Between the Order of S-boxes and the Strength of DES,"
Advances in Cryptology - EUROCRYPT '94 Proceedings, Volume 950 of Lecture
Notes in Computer Science, Springer-Verlag, 1995.
53. Eli Biham, A. Biryukov, "An Improvement of Davies' Attack on DES," CS 817,
EUROCRYPT '94 Proceedings (May 1994), Volume 950 of Lecture Notes in
Computer Science (A. De Santis, ed.), Springer Verlag, 1995, and Journal of
Cryptography, Vol. 10, No. 3, pp. 195-206, 1997.
54. Lars Knudsen, "New potentially weak keys for DES and LOKI," Advances in
Cryptography - EUROCRYPT '94 Proceedings, Volume 950 of Lecture Notes in
Computer Science (A. De Santis, ed.), pp. 419-424. Springer Verlag, 1995.
55. Lars Knudsen, John Erik Mathiassen, "A Chosen-Plaintext Linear Attack on DES,"
Proceedings of Fast Software Encryption 2000, Volume 1978 of Lecture Notes in
Computer Science. Springer-Verlag, 2001.
56. U.S. National Institute of Science and Technology, NIST Special Publication 800-17,
pp. 124 et seq.
57. Bruce Schneier, "Section 17.1 RC4," Applied Cryptography, Second Edition, John
Wiley & Sons, 1996.
58. Anonymous, Subject: RC4 Algorithm revealed, Posting to Usenet newsgroups
sci.crypt, alt.security, comp.security.misc, and alt.privacy, September 14 1994

(reposting of a message to the cipherpunks mailing list). Message-ID:

sternCvKL4B.Hyy@netcom.com

59. Hal Finney, Subject: An RC4 cycle that can't happen, Posting to sci.crypt, 18 September 1994.
60. David Wagner, Subject: Re: Weak Keys in RC4, Posting to sci.crypt, 26 September 1995.
61. Andrew Roos <andrewr@viconix.co.za>, A Class of Weak Keys in the RC4 Stream Cipher, November 1997.
62. John Kelsey, Bruce Schneier, David Wagner, "Key-Schedule Cryptanalysis of 3-WAY, IDEA, G-DES, RC4, SAFER, and Triple-DES".
63. Jovan Golić, "Linear Statistical Weakness of Alleged RC4 Keystream Generator," Advances in Cryptology - EUROCRYPT '97 Proceedings, Volume 1233 of Lecture Notes in Computer Science (W. Fumy, ed.) Springer-Verlag, 1997.
64. Lars Knudsen, Willi Meier, Bart Preneel, Vincent Rijmen, Sven Verdoolaege, "Analysis Methods for (Alleged) RC4," Advances in Cryptology - ASIACRYPT '98 Proceedings, Volume 1514 of Lecture Notes in Computer Science (K. Ohta, D. Pei, eds.), pp. 327-341. Springer-Verlag, 1998.
65. Serge Mister, "Cryptanalysis of RC4-like Ciphers," Master's Thesis, Queen's University, Kingston, Ontario, Canada. May 1998.
66. Serge Mister, Stafford Tavares, "Cryptanalysis of RC4-like Ciphers," Workshop Record of the Workshop on Selected Areas in Cryptography (SAC '98), August 17-18 1998, pp. 136-148.

67. Alexander L. Grosul, Dan S. Wallach, "A Related-Key Cryptanalysis of RC4," Rice University TR00-358, June 2000.
68. Scott Fluhrer, David McGrew, "Statistical Analysis of the Alleged RC4 Keystream Generator," Presented at Fast Software Encryption 2000, New York.
69. Itsik Mantin, Adi Shamir, "A Practical Attack on Broadcast RC4," Presented at Fast Software Encryption 2001.
70. Scott Fluhrer, Itsik Mantin, Adi Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4," In Proceedings of SAC 2001, Eighth Annual Workshop on Selected Areas in Cryptography (Toronto, Ontario, Canada, August 2001), pp. 1-24.
71. Itsik Mantin, "Analysis of the stream cipher RC4," Master's Thesis, Weizmann Insitute, Israel, 2001.
72. G. Durfee, "Distinguishers for the RC4 stream cipher," Manuscript, 2001.
73. Ilya Mironov, "(Not So) Random Shuffles of RC4 (full version)," IACR e-print 2002/067.
74. U.S. National Bureau of Standards (now NIST), "DES Modes of Operation," NBS FIPS PUB 81, U.S. Department of Commerce, December 1980.
75. Bruce Schneier, "Section 9.1 Electronic Codebook Mode," Applied Cryptography, Second Edition, John Wiley & Sons, 1996.
76. U.S. National Bureau of Standards (now NIST), "DES Modes of Operation," NBS FIPS PUB 81, U.S. Department of Commerce, December 1980.
77. Bruce Schneier, "Section 9.3 Cipher Block Chaining Mode,"
78. M. Bellare, A. Desai, E. Jokipii, P. Rogaway, "A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation,"

79. Ron Rivest, Adi Shamir, Leonard Adelman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," MIT Laboratory for Computer Science and Department of Mathematics. Communications of the ACM, February 1978, Volume 21, Number 2, pp. 120-126.
80. PKCS #1: RSA Encryption Standard, An RSA Laboratories Technical Note, Version 1.5. Revised November 1, 1993.
81. Bruce Schneier, "Section 19.3 RSA," Applied Cryptography, Second Edition, John Wiley & Sons, 1996.
82. R. Rivest, A. Shamir, L.M. Adelman, "Cryptographic Communications System and Method," U.S. Patent 4,405,829, filed December 14 1977, issued September 20 1983.
83. Don Coppersmith, Matthew K. Franklin, Jacques Patarin, Michael K. Reiter, "Low-Exponent RSA with Related Messages," Advances in Cryptology - EUROCRYPT '96 Proceedings, Volume 1070 of Lecture Notes in Computer Science (U. Maurer, ed.), pp. 1-9. Springer-Verlag, 1996.
84. Dan Boneh, "Twenty Years of Attacks on the RSA Cryptosystem," Notices of the American Mathematical Society (AMS), Vol. 46, No. 2, pp. 203-213, 1999.

VITA

Graduate College
University of Nevada, Las Vegas

Sourabh Ghose

Local Address:

4213 Chatham Circle Apt#1
Las Vegas, NV 89119

Degrees:

Bachelor of Engineering in Computer Engineering, 2005
University of Bombay, India

Thesis Title: Design, Implementation and Analysis of a Dynamic Cryptography Algorithm with Applications.

Thesis Examination Committee:

Chairperson, Dr. Yoohwan Kim, Ph. D.
Committee Member, Dr. Ajoy Datta, Ph. D.
Committee Member, Dr. Laxmi Gewali, Ph. D.
Graduate Faculty Representative, Dr. Shahram Latifi, Ph. D.